Composition de squelettes algorithmiques : application au prototypage rapide d'applications de vision

Rémi Coudarcher

▶ **To cite this version:**

Rémi Coudarcher. Composition de squelettes algorithmiques : application au prototypage rapide d'applications de vision. Interface homme-machine [cs.HC]. Université Blaise Pascal - Clermont-Ferrand II, 2002. Français. NNT : . tel-00003350

HAL Id: tel-00003350

https://theses.hal.science/tel-00003350

Submitted on 11 Sep 2003

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Introduction

Ces dernières années la puissance des processeurs, et donc des machines séquentielles, a connu un formidable envol. Même les ordinateurs personnels en ont profité, et il n'est pas rare maintenant que la puissance de ces machines atteigne le milliard d'intructions flottantes à la seconde et dépasse le milliard de Hertz en vitesse [VV01] [RS01] [Ler01]. Mais nous avons assisté dans le même temps à une consommation massive de ces ressources par les applications logicielles qui proposent aujourd'hui aux utilisateurs des capacités qui restaient il y a peu encore l'appanage des grands systèmes, aussi bien au niveau de la puissance de calcul que des fonctionnalités ou de l'interface graphique.

Si pour la plupart des applications l'augmentation de la puissance de calcul ne fait qu'améliorer le confort de l'utilisateur (soit en terme de vitesse, soit en terme de fonctionnalités qui lui sont offertes), certaines en sont directement dépendantes.

Parmi celles-ci on trouve notamment les applications dites temps réel, c'est-à-dire des applications qui doivent fournir des résultats dans un temps strictement fixé – en général bref – pour que ces derniers aient un intérêt. Dans ce cas la valeur du résultat seule n'est pas suffisante, il est aussi absolument nécessaire qu'elle soit obtenue en un temps donné pour être par exemple exploitée rapidement (notion de temps réel «application»).

On trouve dans cette classe d'applications des algorithmes de vision, notamment ceux qui sont exploités en robotique mobile ou manufacturière. Ce sont à ces derniers que nous nous adresserons plus particulièrement ici. Pour de tels algorithmes, comme par exemple la détection et le suivi d'objets en mouvement dans une scène routière, le nombre d'opérations à réaliser est très important (de l'ordre du milliard d'opérations élémentaires) et doit être effectué en un temps de l'ordre de la centaine de millisecondes si on veut que le résultat de la perception de l'environnement serve à quelque chose... [AMC+00]

Pour ce type d'applications donc, la puissance de calcul nécessaire est très importante dès lors que l'on veut mettre en œuvre une chaîne d'algorithmes aboutissant à une prise de décision de la part de la machine (par exemple détecter des obstacles dangereux et pouvoir les éviter à temps). La puissance actuelle des processeurs, même si elle est devenue considérable, n'atteint pas encore celle nécessaire pour exploiter sur un seul et unique processeur toute une chaîne d'algorithmes de vision allant du prétraitement de l'image à la prise de décision sur les informations fournies. C'est pourquoi dans ce domaine, comme dans les domaines où la puissance de calcul à mettre en œuvre est très importante (calcul sientifique, météorologie, modélisation de produits aéronautiques,...), il apparaît nécessaire d'utiliser le potentiel qu'offre une machine parallèle.

Les calculateurs parallèles sont apparus pratiquement avec le développement des tout premiers ordinateurs [PH94] [Slo82] [Mac91]. Mais leur utilisation, contrairement aux machines séquentielles, est restée confinée dans des domaines restreints où la puissance de calcul était

primordiale, et n'ont pas subi le même essort, notamment en terme de développement d'outils de programmation, que leurs homologues séquentiels [ST98].

Une des raisons majeures à cet état de fait est sans doute leur difficulté de programmation (coordination des tâches réparties, traitement de plusieurs données simultanément, transfert de ces données entre les tâches, absence de temps global...) [Roy96]. Les modèles classiques, maintenant éprouvés, de programmation des architectures séquentielles sont inadaptés à la programmation des architectures parallèles car ils ne prennent pas en compte ces aspects. Cela tient à la nature même de ces architectures qui mettent en œuvre un réseau d'unités de calcul où les aspects de temporalité, de concurrence, de synchronisme et d'accès aux données sont primordiaux. Seuls quelques modèles récents intégrant la notion d'application distribuée tentent d'y remédier. Mais là encore la programmation se fait plus en faisant collaborer des unités de programmes qu'en intégrant complètement la notion de parallélisme tout au long du développement et notamment au sein même d'un algorithme.

De plus, on constate la très grande difficulté à penser un problème, et donc son algorithme de résolution, sous forme parallèle, c'est-à-dire où une partie de la résolution se ferait de manière concurrente. Force est de constater que la plupart des problèmes ont été formulés pour être traités séquentiellement, étape par étape, et non en essayant de dégager des phases où plusieurs étapes de travail dans la résolution pourraient être réalisées simultanément [Fly96]. Cet aspect des choses tient probablement au fait qu'on pense encore, malgré tout, les algorithmes comme s'ils devaient être exécutés par l'homme, et qui plus est un seul homme ! Le programmeur ne se préoccupe généralement de la parallélisation de son algorithme que lorsque le temps d'exécution de celui-ci devient prohibitif [SSS98].

Quelques méthodologies et outils sont apparus pour aider les programmeurs dans cette lourde tâche qu'est la programmation d'une machine parallèle (pour la plupart issus du milieu universitaire) [Che93] [ST98], mais aucun n'a atteint la maturité des outils proposés pour les architectures séquentielles. Une des difficultés est qu'il existe encore à l'heure actuelle de nombreux types différents d'architectures parallèles. La mise en œuvre d'outils d'aide à la programmation implique donc de pouvoir proposer au niveau du programmeur un certain niveau d'abstraction vis-à-vis de l'architecture sous-jacente, et cela afin que l'outil puisse être utilisé sur de multiples plate-formes, aucune architecture parallèle ne semblant se dégager du lot pour devenir l'architecture universelle.

Le travail que nous présentons ici concerne les outils d'aide à la programmation parallèle de machines de type «Multiple Instruction stream Mutliple Data stream - Distributed Memory», ou MIMD-DM, selon la classification de M.J. Flynn [Fly66]. Nous nous intéresserons tout particulièrement à l'application de ces outils dans le cadre de la programmation d'applications de vision artificielle.

L'outil qui est présenté dans ce mémoire, appelé SKiPPER pour SKeleton-based Parallel Programming EnviRonment, est un outil d'aide à la programmation parallèle pour le prototypage rapide d'applications de vision artificielle. D'une manière générale **le but du prototypage rapide dans le cadre du génie logiciel est *l'obtention d'une application opérationnelle dans un délai très court et avec le minimum d'étapes intermédiaires dans le processus de réalisation***. Dans notre cas le prototypage rapide n'aura de sens que lors de la phase d'implantation sous forme parallèle d'un algorithme de vision déjà opérationnel sous forme séquentielle.

Comme tout outil d'aide à la programmation parallèle, il doit être en mesure de faciliter le passage d'un algorithme séquentiel à sa version parallèle. Pour ce faire, il doit donc limiter le nombre d'étapes dans le processus de conversion, réduire la connaissance de la machine que

doit avoir le programmeur, et éviter l'utilisation de fonctions spécifiques (comme par exemple l'utilisation de bibliothèques de communication auquel un programmeur de machines séquentielles est peu, voire pas, habitué).

La méthodologie retenue pour la mise en œuvre de SKiPPER est fondée sur l'emploi de *squelettes algorithmiques*. Ces structures de programmation paramétrables ont été initialement formalisées par Cole [Col89] et Skillicorn [Ski90]. Ce sont des structures de programmation qui encapsulent des formes communément utilisées de parallélisme pour les rendre facilement exploitables. Un squelette représente un schéma de calcul/communication caractéristique qui peut être instancié par des fonctions de calcul spécifiques fournies par le programmeur afin de former l'application. Le travail décrit ici s'intéresse à l'aspect compositionnel de ces squelettes afin d'augmenter le potentiel d'applications de vision artificielle qui peuvent être spécifiées avec notre outil. Le séquencement de squelettes étant un cas trivial de composition déjà pris en charge dans les premiers développements de SKiPPER, nous nous intéressons ici au cas plus difficile de l'imbrication de plusieurs squelettes, imbrications homogène et hétérogène, c'est-à-dire entre squelettes identiques ou non. L'imbrication de deux squelettes est la capacité qu'a l'un d'eux à pouvoir accepter l'autre comme «fonction de calcul», c'est-à-dire que les traitements du premier sont parallélisés par le second qui réalise effectivement les opérations.

Le mémoire se décompose donc comme suit.

Le premier chapitre traite du contexte de l'étude et plus particulièrement de la problématique posée par le prototypage rapide d'applications parallèles en vision artificielle, ainsi que des squelettes algorithmiques. Nous verrons que l'utilisation des squelettes algorithmiques peut constituer une réponse satisfaisante au problème du prototypage rapide.

Le second chapitre est dédié à la présentation de la première mouture de l'environnement de programmation parallèle SKiPPER-I qui nous a servi de substrat pour l'étude de la composition des squelettes. Cette environnement s'appuie sur l'utilisation des squelettes algorithmiques et a été conçu en vue du prototypage rapide d'applications de vision artificielle. Les principes qui sous-tendent SKiPPER y sont présentés, ainsi que les différentes possibilités de composition des squelettes. Les limitations en terme de composition, ajoutées au fait que les squelettes dynamiques de la bibliothèque étaient difficiles à représenter dans le modèle d'exécution statique de SKiPPER-I, ont conduit au développement d'une nouvelle version présentée au chapitre quatre.

Le troisième chapitre présente un état de l'art des environnements de programmation parallèle autorisant la composition de squelettes, notamment dans le sens de l'imbrication. La plupart de ces environnements sont issus de la recherche universitaire internationale. De tels projets sont en effet développés en Italie, en Ecosse ou bien encore au Canada.

Le chapitre quatre aborde alors en détail la nouvelle version de SKiPPER développée dans le cadre de cette thèse (SKiPPER-II), et qui prend en compte les aspects de composition des squelettes évoqués plus haut. L'intérêt d'une telle démarche est d'augmenter les capacités de parallélisation potentielle des applications de vision artificielle. Dans ce cadre, nous présentons les nouveaux aspects et mécanismes qui ont présidé à la réalisation de la nouvelle version du noyau de SKiPPER (K/II), notamment la mise en œuvre d'un modèle d'exécution unique pour tous les squelettes.

Enfin le chapitre cinq présente les expérimentations que nous avons réalisées avec SKiPPER-II. Trois algorithmes de traitement d'images sont dans un premier temps présentés (calcul d'histogramme, détection de taches lumineuses dans une image et division récursive d'images) ne mettant en œuvre qu'un seul squelette à chaque fois (Split-Compute-Merge, Data Farming et

Task Farming respectivement). L'intérêt est ici de comparer les résultats avec la première version de SKiPPER en validant la reprise sans modification de ces algorithmes déjà utilisés pour l'évaluation de cette dernière. Dans un deuxième temps sont présentés deux algorithmes avec imbrication de squelettes. Le premier est une application «synthétique» qui est comparée à sa version écrite à la main. Le second est un produit matriciel sur des matrices d'entiers de longueurs arbitraires. Enfin une application complète de vision artificielle nécessitant l'imbrication de squelettes pour sa parallélisation est décrite.

Chapitre 1

Prototypage rapide et squelettes algorithmiques

Nous abordons dans ce chapitre les notions de prototypage rapide logiciel et de squelettes algorithmiques. Nous présentons le prototypage rapide comme une méthode de développement bien adaptée à la programmation parallèle, notamment pour des applications de vision artificielle. L'utilisation des squelettes algorithmiques constitue une forme de mise en œuvre efficace du prototypage rapide.

1.1 Introduction

Les récentes avancées technologiques auquelles nous avons assisté ces dernières années ont permi la conception de machines parallèles suffisament performantes pour fournir des solutions satisfaisantes, notamment en terme de temps d'exécution, à des problèmes issus des domaines du calcul scientifique, du contrôle en temps réel, de la simulation ou bien encore des bases de données (voir figure 1.1). De plus la chute des coûts de production en ont fait des machines de plus en plus accessibles (notamment avec l'émergence des *grappes d'ordinateurs personnels*, et tout particulièrement la classe des machines *Beowulf* [BM01] (voir figure 1.2)).

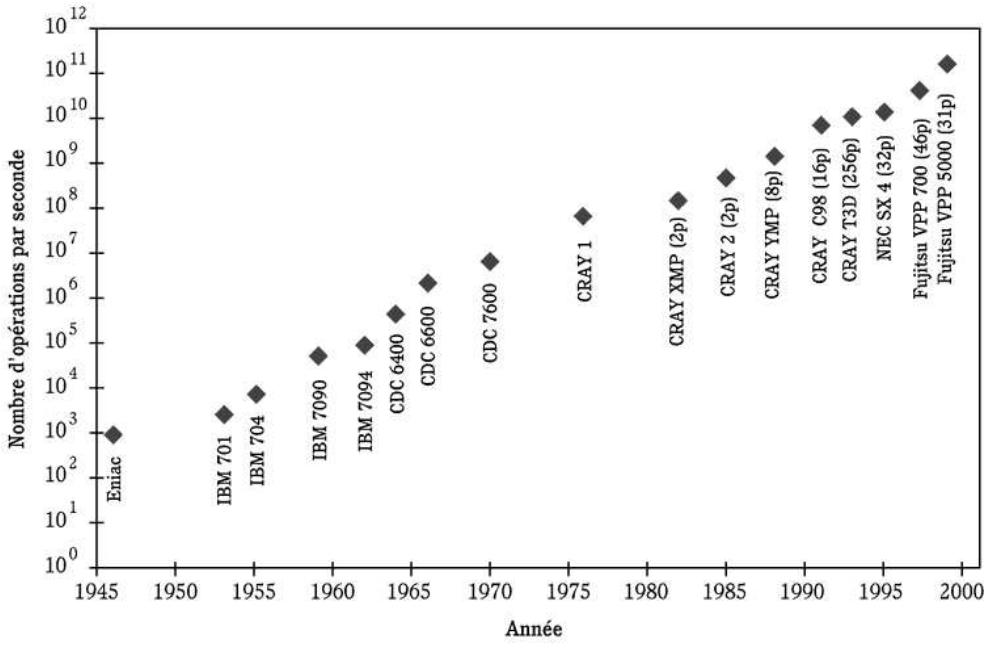


FIG. 1.1 – *Evolution de la puissance de calcul des grands ordinateurs scientifiques (Source : Météo-France).*

Cependant les applications sachant tirer un réel profit de ces architectures ne se sont pas développées au même rythme. Elles ont par ailleurs montré leur difficulté de développement et plus encore de maintenabilité, sans pour autant atteindre les performances souhaités par leurs concepteurs.

A. Goldberg *et coll.* [GMN+94] mettent en avant le manque d'outils de développement et de langages de programmation pour les applications parallèles qui soient suffisament indépendants de l'architecture de la machine utilisée, alors que les calculateurs parallèles eux-mêmes se développent rapidement mais sans que s'impose aucune architecture. Le principal problème rencontré est la différence souvent importante qui peut exister entre le modèle de parallélisation utilisé pour concevoir un algorithme et celui proposé par la machine cible [NP92] ; le parallélisme potentiel de l'application ne correspond pas toujours au parallélisme effectif de l'architecture exploitée. Au delà de cette vision pragmatique des choses, D.B. Skillicorn et D. Talia, menant dans [ST98] une revue de modèles et langages de programmation parallèle pouvant renverser la tendance, donnent un certain nombre d'éléments pertinents concourrants à cet état de fait.

RANK	MANU-FACTURER	COMPUTER	GF/S	INSTALLATION SITE	COUNTRY	YEAR	AREA OF INSTALLATION	# PROC
33	Sun	HPC 450 Cluster	272.1	Sun, Burlington	USA	1999	Vendor	720
34	Compaq	Alpha Server SC	271.4	Compaq Computer Corp. Littleton	USA	1999	Vendor	512
...	...	...	...	...	...	...	...	...
44	Self-made	Cplant Cluster	232.6	Sandia National Laboratories	USA	1999	Research	580
...	...	...	...	...	...	...	...	...
169	Self-made	Alphleet Cluster	61.3	Institute of Physical and Chemical Res. (RIKEN)	Japan	1999	Research	140
...	...	...	...	...	...	...	...	...
265	Self-made	Avalon Cluster	48.6	Los Alamos National Lab/ CNLS	USA	1998	Research	140
...	...	...	...	...	...	...	...	...
351	Siemens	hpcLine Cluster	41.45	Universitaet Paderborn/PC2	Germany	1999	Academic	192
...	...	...	...	...	...	...	...	...
454	Self-made	Parnass2 Cluster	34.23	University of Bonn/ Applied Mathematic	Germany	1999	Academic	128

FIG. 1.2 – *Performances des machines Beowulf classées au TOP500 (Source : TOP500 Supercomputers).*

Le premier, comme nous l'avions mentionné en introduction, est la difficulté de la pensée humaine à développer une solution parallèle à un problème. Malgré l'apparent aspect parallèle du monde et tout particulièrement de la nature intime de la pensée, le développement d'algorithmes de leur conception à leur mise au point reste profondemment séquentiel. Cela rend d'autant plus difficile l'accès à une programmation parallèle de même niveau que la programmation séquentielle.

Le second qu'ils mettent en avant rejoint l'argumentation d'A. Goldberg *et coll.* en disant que les techniques ne programmation parallèles ne se sont développées qu'après la technologie parallèle. Elle en est la résultante, et donc à ce titre un simple moyen d'usage de l'architecture. Elle n'a pas encore atteint le niveau de maturité lui permettant de s'abstraire de son support pour que soit développée une véritable théorie de la programmation parallèle.

Qui plus est, et même en restant au niveau de l'exploitation directe de l'architecture parallèle, et c'est le troisième point, il est très délicat de déterminer sur quels facteurs jouer pour améliorer les performances d'un algorithme. En effet, il existe une interaction subtile entre tous les éléments du système, que ce soit au niveau processeur, mémoire ou réseau de communication, qui fait que l'amélioration des performances au niveau d'un des nœuds de calcul ne signifie pas forcément une amélioration des performances de l'ensemble ; les paramètres doivent être considérés dans leur intégralité. Et comme le soulignent A. Goldberg *et coll.*, les différences architecturales n'améliorent pas les choses. D.B. Skillicorn et D. Talia rappellent que, si l'efficacité d'un programme séquentiel ne varie pas plus que d'un facteur constant lors de son exécution sur des processeurs différents, il n'en va pas de même pour un programme parallèle. La nature «non-locale» des algorithmes parallèles fait que le réseau de communication joue un rôle significatif dans les performances globales. La portabilité d'un algorithme en est aussi la conséquence et, comme nous le verrons par la suite, est une caractéristique importante pour la diffusion de la programmation parallèle. Le portage d'un algorithme d'une architecture à une autre reste encore une entreprise demandant beaucoup de travail.

La nécessité de langages, ou d'environnements, de programmation parallèle de haut niveau se fait clairement sentir par le fait que le programmeur d'applications s'intéresse prioritairement à la résolution de son problème, plutôt qu'aux difficultés techniques que peut engendrer l'emploi de telle ou telle architecture parallèle. Les langages de programmation parallèle faisant intervenir explicitement des instructions liées à la nature même de la machine utilisée souffrent d'un manque évident de portabilité, mais aussi de souplesse pour permettre au programmeur d'explorer un vaste champ de possibilités de conception de son algorithme. On est alors confrontés à la définition d'un modèle de programmation (au sens donné par D.B. Skillicorn et D. Talia [ST98]) offrant un compromis «acceptable» entre abstraction et efficacité.

L'élaboration d'environnements de programmation parallèle exploitant le concept de prototypage rapide tente de répondre à ces problèmes.

Dans la suite de ce chapitre nous présentons brièvement les solutions «classiques» au problème posé par la programmation parallèle à travers des techniques telles que bibliothèques et langages spécialisés. Suite à cela, nous présentons le prototypage rapide comme une réponse à ce problème, en présentant les approches les plus significatives retenues dans ce contexte. Dans le cadre du prototypage rapide comme solution de programmation parallèle, nous envisageons l'apport des méthodologies fondées sur les squelettes algorithmiques. Enfin, nous proposons un rapide survol de réalisations concernant le domaine de la vision artificielle.

1.2 Outils et langages de programmation parallèle

Pour programmer un algorithme de manière parallèle, on peut faire appel à trois grands groupes de techniques :

- des bibliothèques de fonctions spécialisées,
- des interfaces de communication,
- des langages spécialisés.

1.2.1 Les bibliothèques de fonctions spécialisées

1.2.1.1 Principes

Par bibliothèques de fonctions spécialisées nous entendons des bibliothèques écrites pour résoudre une classe de problèmes donnés dans un domaine particulier comme ceux du calcul scientifique, de l'aéronautique, de la biologie, *etc*. La bibliothèque PBLAS (Parallel Basic Linear Algebra Subprogram) [DCDH90] par exemple implémente, dans le domaine du calcul scientifique, des outils d'algèbre linéaire. L'utilisateur (le programmeur d'application parallèle) a à sa disposition un ensemble figé de fonctions dont l'objet est strictement délimité par le domaine d'application. Ces fonctions ont été écrites par des spécialistes du parallélisme et du domaine considéré afin de tirer profit au maximum d'un certain nombre d'architectures pour résoudre les problèmes auquels la bibliothèque s'adresse. L'utilisateur peut alors raisonnablement considérer qu'il pourra, par l'intermédiaire de ces fonctions, exploiter au mieux le parallélisme de sa machine (si tant est que cette machine fasse partie de celles ciblées par la bibliothèque).

1.2.1.2 Exemples de réalisations

De telles bibliothèques ont été développées dans de nombreux domaines scientifiques et notamment celui qui nous intéresse : le traitement d'images. Notamment, l'équipe ISIS (Intelligent Sensory Information Systems) de l'université d'Amsterdam aux Pays-Bas a produit une bibliothèque parallèle pour le traitement d'images sur machine MIMD à mémoire distribuée [SKG00] [SK00b]. Conformément à ce que nous venons d'exposer, le programmeur n'a pas besoin d'avoir de notions en parallélisme pour l'utiliser. Cette bibliothèque contient tout un ensemble de fonctions de traitement d'images bas niveau identifiées comme communément employée dans la communauté scientifique correspondante. L'intérêt de ce genre de bibliothèque est qu'elle présente des prototypes de fonctions rigoureusement identiques à ceux de fonctions similaires mais s'exécutant en séquentiel. L'utilisateur n'est donc pas perturbé par la parallélisation de l'algorithme de traitement d'images qui a été fait, et utilise la version parallèle comme il utiliserait la version séquentielle. De plus, le langage choisi pour supporter l'implantation est le langage C/C++, reconnu pour être très employé pour ce type d'algorithmes. Au sein de cette architecture logicielle, les différentes fonctions parallélisées le sont, non pas en essayant d'optmiser leur code pour telle ou telle machine cible, mais en fournissant un code optimum pour un modèle d'architecture formant une machine virtuelle. Ainsi les fonctions de la bibliothèques sont écrites pour cette machine virtuelle et non pas pour une machine cible physique donnée, ni même pour un ensemble de machines cibles. Le portage s'en trouve facilité. Afin de ne pas faire chuter les performances lors d'un changement d'architecture cible, la machine virtuelle est dotée d'un modèle de performance [SK00a]. Chaque opération qui peut être réalisée sur cette machine virtuelle est accompagnée d'un micro-modèle de performance. Ce dernier est paramètré par un grand nombre de valeurs, dont notamment le type de donnée instanciée et des indicateurs de charge [SKG00]. Etant donné la combinatoire induite par le grand nombre de paramètres, la mise en correspondance de la machine virtuelle et de la machine cible réelle est établie par une heuristique guidée par le modèle d'exécution. Elle peut être raffinée, à la demande de l'utilisateur, par la prise en compte des résultats des outils d'analyse de performances (fournis avec la bibliothèque) sur l'exécution du programmeur de l'utilisateur.

Une autre implémentation notable d'une bibliotèque de traitement d'images a été faite au MIT dans le cadre de recherches en Astronomie [Kep01]. Cette bibliothèque comme la précédente propose à l'utilisateur un certain nombre de fonctions de traitement d'images parallélisées sans qu'il n'ai besoin de connaître la façon dont cela a été réalisé. Cette bibliothèque s'appuie sur des standards aussi bien pour le traitement d'images avec l'emploi de la bibliothèque VSIPL (Vector, Signal and Image Processing Library), que pour la parallélisation avec OpenMP [Boa01]. Les langages de programmation pouvant être utlisés sont le langage C et le Fortran. Dans ce cadre leurs machines cibles sont toutes les machines disposant de ces deux standards (SGI, HP, Sun, IBM et les systèmes Linux).

1.2.1.3 Avantages et inconvénients

Une telle approche offre des avantages certains pour l'utilisateur :

- facilité d'exploitation du parallélisme puisque totalement transparent pour lui,
- quasi-certitude d'une exploitation optimale du parallélisme offert par sa machine,
- développement de ses algorithmes indépendamment de la nature même (parallèle ou non) de la machine cible.

Mais elle a aussi des inconvénients, inconvénients qui sont le pendant direct des avantages qu'elle propose :

- le parallélisme ne peut être exploité que lors des appels aux fonctions de la bibliothèque,
- manque de souplesse pour l'écriture des algorithmes lorsque ceux-ci s'éloignent un peu trop de la classe de problèmes traités par la bibliothèque (cette technique est dite «dépendante de l'application»),
- l'utilisateur ne peut pas choisir entre plusieurs possibilités de parallélisation d'un même algorithme ; il est condamné à utiliser celui qui lui est imposé, sans même savoir quel type de parallélisation est utilisée (ce qui est un avantage pour la plupart des utilisateurs qui ne souhaitent pas s'y intéresser, mais qui peut devenir un inconvénient si la machine qu'il utilise n'était pas explicitement ciblée par la bibliothèque).

Ce dernier point est notamment important pour les applications qui, tenant compte de la différence entre la taille du problème à traiter et la taille de la machine (en terme de nombre de processeurs), font appel à des algorithmes différents pour la résolution en fonction des ressources disponibles.

Son manque de souplesse pour la création d'applications parallèles fait que ces bibliothèques sont rarement utilisées seules, mais en conjoinction avec des langages qui offrent des extensions permettant la prise en compte du parallélisme.

1.2.2 Interfaces de communication

Sous ce vocable on peut regrouper toutes les extensions de langages déjà existants pour la programmation séquentielle et qui permettent l'accès aux fonctionnalités propres des machines parallèles, essentiellement tous les mécanismes de communication qu'elles peuvent offrir.

Les deux plus connues sont MPI (Message-Passing Interface) [For94] [GLS94] [MMHB96] et PVM (Parallel Virtual Machine) [Sun90] [GBD+94]. L'utilisation de ces interfaces est devenue très courante. Elles sont aussi simple d'emploi dans le sens où l'utilisateur n'a pas à apprendre un nouveau langage de programmation : leur mise en œuvre se fait directement dans la syntaxe du langage hôte (C, Fortran,...). Leur grand avantage est de laisser une totale liberté de programmation à l'utilisateur en lui offrant toutes les possiblités que lui autorise sa machine (parfois même un peu plus par émulation de certains schémas de communication). Le programmeur peut alors complétement spécifier, dans ses moindres détails, le schéma de parallélisation qu'il souhaite utiliser.

L'inconvénient majeur de cette approche est qu'elle est dépendante de l'architecture matérielle sous-jacente. En effet, si le schéma de parallélisation choisi pour un algorithme peut rester indépendant de la topologie du réseau d'interconnexion utilisé, le programmeur soucieux de performances et connaissant l'architecture de la machine cible choisira préférentiellement un schéma dont le «mapping» sur le réseau est le plus direct ; or ce schéma est alors de fait explicitement écrit dans le corps de l'algorithme par l'emploi d'instructions faisant appel aux fonctions de l'interface (Send, Receive,...). Avec une telle approche il faut subtilement choisir le bon équilibre entre performances (qui nécessite un schéma adapté à la topologie) et maintenabilité (qui suggère l'emploi de schémas généraux). La liberté de programmation se paye par une plus grande difficulté, non seulement de programmation de l'application car il faut réfléchir à la fois au schéma de parallélisation que l'on souhaite utiliser et à sa mise en œuvre, mais aussi de maintenabilité de cette application. L'entrelacement des instructions purement de résolution

du problème traité avec celles permettant de le paralléliser (instructions de communication et de synchronisation pour l'essentiel) détériore la lisibilité de l'algorithme.

Ce dernier point n'a pas seulement une influence sur la durée de vie de l'application en facilitant ou non son maintien dans le temps, mais a aussi une influence négative sur sa mise au point et son amélioration. En effet, tout changement dans le schéma de parallélisation, voire dans l'algorithme de résolution, peut imposer de revoir l'ensemble du code, algorithme de résolution et instructions de parallélisation, les deux étant alors fortement liés ; or dans beaucoup de cas, la possibilité de faire évoluer une application est une caractéristique essentielle.

Comme le souligne L.S. Nyland [NP92], proposer au programmeur la possibilité d'intervenir sur tous les détails de l'implantation rend le développement d'une application délicat et l'expérimentation de nouvelles voies algorithmiques lourde à réaliser. La portabilité de l'application s'en trouve aussi amoindrie et donc son utilité éventuelle car elle ne peut pas forcément être exploitée sur plusieurs machines.

Outre les interfaces de communication par «passage de messages», se trouve aussi un autre modèle de parallélisme, de plus en plus exploité dans les architectures commerciales, qui est le modèle de mémoire partagée. Des interfaces spécifiques ont donc été développées pour lui comme OpenMP [Boa01].

Ce modèle convient aussi aux langages de programmation de haut niveau. La machine parallèle supposée être exploitée est composée de plusieurs processeurs dont l'espace mémoire est commun. En fait, ce modèle n'impose pas forcément que l'architecture dispose physiquement d'une mémoire partagée. Ce type de mémoire peut très bien être simulée par des techniques de passage de messages sur des architectures à mémoire distribuée (on parle alors de machine à mémoire *virtuellement partagée*, ou DSM pour Distributed Shared Memory). Dans tous les cas, les processeurs exécutent leurs instructions avec des mécanismes de synchronisation permettant d'éviter les conflits d'accès à la mémoire par verrous ou sémaphores par exemple. L'un des avantages de ce modèle est sa souplesse d'utilisation. Les variables peuvent être facilement partagées entre des processus s'exécutant sur des processeurs différents sans requérir de la part du programmeur la mise en œuvre d'instructions spécifiques. Le partage est fait de manière transparente. Le revers est de rester suffisamment vigilant sur l'utilisation de ressources partagées pour prendre en compte les latences d'accès à la mémoire et le «goulot d'étranglement» qu'elle peut engendrer.

Il faut enfin noter que l'écriture de programmes suivant ce modèle permet d'obtenir du code portable sur différentes plates-formes architecturées autour d'une mémoire partagée et, dans une certaine mesure, avec des performances prévisibles lorsque les ressources en unités de calcul augmentent et que la mémoire est physiquement partagée (modélisation de la gestion des bus, hiérarchie de mémoires). Cependant, il est irréaliste de considérer un coût uniforme d'accés à la mémoire, surtout si le nombre de processeurs est important et que la quantité de mémoire qui est partagée l'est aussi.

Les travaux que nous présentons dans ce mémoire (chapitres 2 et 4) concernent exclusivement les machines à mémoire *physiquement distribuée*.

1.2.3 Les langages spécialisés

Ces langages sont conçu pour la programmation parallèle spécifiquement (SR [AOC+88], ORCA [BKT92], Nesl [BHS+93], TwoL [RR96], Fork95 [KS97], NestStep [Keb99]), afin de prendre en charge de manière native tous ses aspects. Leur utilisation est une autre approche de la programmation parallèle. Si elle offre certains avantages comme une forte homogénéité dans la programmation puisque le langage utilisé est dédié à la programmation parallèle, ou encore comme l'exploitation maximale d'une machine donnée si le langage a été spécialement développé pour, elle présente aussi des inconvénients.

Le problème majeur de cette approche est que l'utilisateur est obligé d'apprendre entièrement un nouveau langage de programmation. Or, bien souvent, il n'a pas de temps pour cela. Son objectif premier étant de porter ses algorithmes déjà existant sur une machine parallèle pour obtenir des performances accrues. De plus, le problème s'étend à la capacité de réutilisation du code déjà écrit dans des langages «conventionels». L'utilisation de nouveaux langages de programmation oblige l'utilisateur à réécrire l'ensemble de son code, là où il ne souhaiterait que de s'occuper de la mise en œuvre du parallélisme.

1.3 Le prototypage rapide comme méthode de programmation parallèle

Avec la mise sur le marché de calculateurs parallèles de plus en plus puissants et l'intérêt grandissant pour le parallélisme et le calcul distribué, ont émergé un certain nombre de modèles de programmation, d'outils de développement et de systèmes. Les utilisateurs font donc face maintenant au problème délicat de trouver le bon outil et la bonne méthode de programmation parallèle, celle qui correspondra le mieux aux problèmes qu'ils se posent. Ce choix est fonction d'un grand nombre de paramètres dont certains dépendent directement de l'utilisateur lui-même où de l'environnement de programmation dont il dispose déjà. Par exemple, la plupart des programmeurs demandent de conserver un langage de programmation auquel ils sont habitués comme le C ou le Fortran. Un autre point important est la possibilité offerte ou non de réutiliser au maximum le code déjà écrit.

Le prototypage rapide et ses outils associés offrent une méthode de programmation qui facilite le travail du programmeur qui veut porter une application, déjà écrite pour une machine séquentielle, sur une machine parallèle pour tirer profit de sa puissance. Cette approche tente de proposer un moyen pour passer de la version séquentielle de l'application à une description formelle de la version parallèle de l'algorithme. Cela est destiné à procéder, de la manière la plus automatique possible, à son implantation en tenant compte des moyens de parallélisation existants.

1.3.1 Définition

D'une manière générale (et en dehors de toute considération sur le parallélisme), l'objet du prototypage rapide logiciel est la conception d'une application dans un délai relativement court (par rapport au temps de développement de cette classe d'applications), avec un minimum d'efforts et en un minimum d'étapes. Nous pouvons citer à ce propos une étude menée en 1993 [SS94]. Les auteurs de cette expérience ont demandé à des étudiants d'écrire un programme parallèle pour résoudre un problème mathématique. Une moitié des étudiants ont dû le faire en

utilisant une interface de type PVM, l'autre moitié en utilisant un environnement de développement parallèle faisant appel au prototypage rapide (*Entreprise* [SSS98]). Le travail des étudiants était contrôlé pour obtenir des données statistiques sur le nombre de compilations, de tentatives d'exécutions, d'édition du code source et sur le temps passé. Le nombre de lignes de code dont ils ont eu besoin pour écrire leur programme ainsi que l'accélération qu'ils ont obtenus ont été consignés. Il en est resorti que les étudiants utilisant l'outil *Enterprise* passaient moins de temps à développer leurs algorithmes, mais leurs solutions n'étaient pas aussi performantes que celles obtenus par une programmation directe. L'utilisation d'*Enterprise* permettait au étudiants d'économiser l'écriture de 66 % de lignes de code, et ils devaient faire moins d'éditions, de compilations et de tests pour obtenir un programme opérationnel. Etant donné la relative facilité de mise en œuvre du parallélisme dans cet environnement, il est apparu que les étudiants qui l'utilisaient passaient plus de temps que les autres à essayer d'améliorer les performances de leurs algorithmes. Une autre étude menée en 1995 [SS96] est venue confirmer ces résultats, mais aussi montrer que des étudiants qui avaient déjà programmé avec des outils comme PVM avant d'utiliser un environnement comme Enterprise restaient insatisfaits des possibilités offertes en raison d'un contrôle moindre sur les possibilités de mise en œuvre de schémas de parallélisation très particuliers. Appliqué à la programmation parallèle, le prototypage rapide aura pour but de réduire considérablement le temps nécessaire à l'élaboration d'une version parallèle d'un algorithme séquentiel déjà existant en réduisant le travail que devra fournir l'utilisateur pour y parvenir (acquisition des notions, techniques et langages relatifs au parallélisme et implémentation de ceux-ci). On désire alors produire une application qui permettra d'évaluer les aspects les plus critiques du logiciel, en termes notamment d'efficacité de certains algorithmes, avant d'en faire le produit final. En effet, le choix de passer à une version parallèle d'un algorithme provient souvent de la nécessité d'une plus grande puissance de calcul. Une application donnée présente des sections de code où les aspects temporels sont plus critiques que dans d'autres et ce sont celles-ci qui devront bénéficier du plus d'attention au moment de la parallélisation. De ce fait, dans une première phase de développement, seules les sections concernées peuvent faire l'objet d'un prototype parallèle, les autres venant se greffer ultérieurement pour former l'application finale.

J. Hoffman et J. Margerum-Leys [HML96] présentent le prototypage rapide comme une méthode exploitant un cycle de conception en spirale (de type incrémental) :

1. spécification de l'application,
2. implémentation à l'aide d'un outil de prototypage,
3. évaluation des performances du prototype,
4. si nécessaire, correction de l'algorithme et retour en 2.

On peut noter que la plus grande différence avec des modèles plus traditionnels de programmation est que ceux-ci mettent l'accent sur une conception strictement définie de l'algorithme dès le tout début du cycle de développement, alors que le prototypage rapide suit une voie plus pragmatique en imposant à chaque étape de la conception le minimum de contraintes nécessaires pour atteindre les objectifs fixés. Le processus est intrinséquement incrémental. A chaque itération l'algorithme est amélioré jusqu'à atteindre les performances souhaitées. Cette méthode convient parfaitement à la transformation d'algorithmes séquentiels en algorithmes parallèles lorsque le schéma de parallélisation optimum n'est pas connu *a priori*. On cherche alors en affinant l'idée de départ par essais et corrections à obtenir le schéma adéquat.

1.3.2 Quelques approches du prototypage rapide pour le parallélisme

Pour exploiter cette méthode de programmation, il faut concevoir des outils où l'utilisateur n'ait pas à s'occuper de la manière dont peut être réalisé le ou les schémas de parallélisation qu'il choisira. Cela est notamment une des clés qui lui permettent, en pouvant changer rapidement de schéma, d'évaluer facilement le comportement de son algorithme avec diverses approches parallèles.

Nous présentons ici deux approches significatives du problème, en dehors de celle fondée sur les squelettes que nous détaillons à la suite.

1.3.2.1 Méthodologie Adéquation Algorithme Architecture - AAA

Cette méthodologie a été formalisée et développée à l'INRIA [1] dans le cadre du projet SOSSO, sur les applications et les outils de l'automatique, à Rocquencourt.

Elle consiste à étudier simultanément les aspects algorithmiques et architecturaux, ainsi que leurs interactions, dans le but d'effectuer une implantation efficace de l'algorithme sur l'architecture, tout en satisfaisant certaines contraintes exécutives, notamment de temps réel [Sor94] [Sor96] (voir figure 1.3).

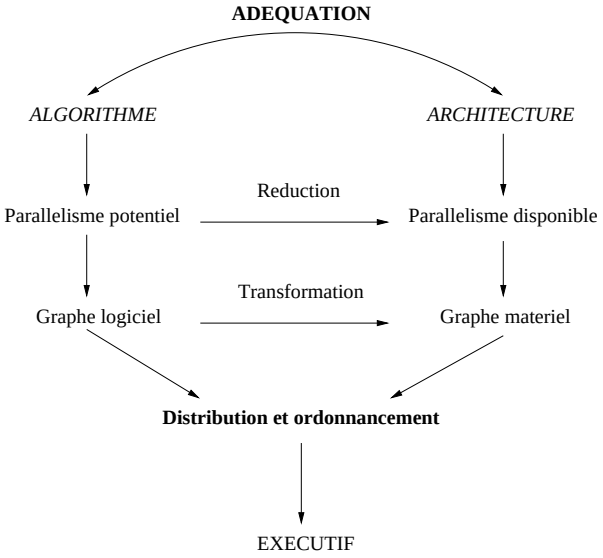


FIG. 1.3 – *La méthodologie AAA.*

Cette méthodologie repose sur deux modèles distincts : l'un représente le logiciel et l'autre le matériel. Ainsi la spécification de l'application et la spécification de son implantation pour une machine cible donnée sont clairement dissociés. On peut remarquer que cette distinction procède d'ores et déjà d'une volonté de simplifier le portage et la conception des algorithmes.

La partie logicielle est modélisée sous la forme d'un graphe flot de données conditionné (GFDC). Ce type de représentation induit un ordre partiel sur les opérations. Cela permet d'exprimer plus facilement le parallélisme potentiel dans la mesure où, si deux somets du graphe n'ont pas de dépendance de données mutuelle, alors il sont potentiellement exécutables en parallèle.

1. INRIA : Institut National de Recherche en Informatique et en Automatique.

La partie matérielle est quant à elle modélisée par un graphe non orienté dans lequel chaque sommet représente un processeur, et chaque arc un lien de communication entre deux processeurs.

Le terme d'*adéquation* signifie donc, dans ce contexte, la mise en correspondance (efficace) des deux graphes précédents. Il est procédé à une réduction du parallélisme potentiel au parallélisme effectif. Elle est obtenue par transformation du GFDC pour le faire correspondre au graphe de la partie matérielle. Cette transformation inclut l'ensemble des distributions (allocations spatiales) et des ordonnancements (allocations temporelles) nécessaires. Le problème résultant de cette mise en correspodance étant réputé NP-complet, un certain nombre d'heuristiques sont employées pour sélectionner la solution «optimale» dans l'espace des solutions.

Conformément à cette méthodologie, l'outil SynDEx [GLS98] [GLS99] [Sor02] a été développé. Il répond de fait aux critères énoncés plus haut pour le prototypage rapide :

- réduction des erreurs de développement par l'utilisation d'un modèle d'architecture indépendant du modèle de l'algorithme,
- phase de placement et d'ordonnancement automatisée et rapide,
- programmation «système» déléguée entièrement à l'outil.

On notera qu'avec cette approche le programmeur doit spécifier explicitement le parallélisme de son application. Il ne s'agit donc pas d'une approche «tout automatique» («tout implicite», au sens de [ST98]).

1.3.2.2 Méthodologie de type *Design Patterns*

La notion de *Design Pattern* a été inventée par Christopher Alexander, architecte dans le civil, en 1977 [Ale77] [Ale79]. Elle est définie comme «une solution à un problème dans un contexte». Le *contexte* fait référence à un ensemble de situations récurrentes dans lesquelles le motif s'applique. Le *problème* se réfère à un ensemble de forces, objectifs et contraintes, qui ont lieu dans ce contexte. La *solution* finalement est une règle de conception qu'on peut appliquer pour résoudre ces forces. Ce concept particulèrement riche possède beaucoup de domaines d'application [GHJV94]. Les *Design Patterns* permettent de capitaliser l'expérience des concepteurs les plus expérimentés et ainsi d'éviter de «réinventer la roue». Qui plus est, un concepteur familiarisé avec ces motifs peut les appliquer immédiatement sans avoir à les redécouvrir et communiquer plus facilement sur ses conceptions (les motifs deviennent ainsi un vocabulaire de référence commun entre programmeurs).

Si au tout départ de leur existence les *Design Patterns* n'étaient considérés que dans le cadre de la programmation séquentielle pour les domaines liés à l'informatique, de récentes recherches ont proposés d'appliquer ce concept à la programmation parallèle [MMS00] [MSSB00] [Dan01]. De fait, les *Design Patterns* parallèles sont présentés comme des abstractions modélisant des schémas de parallélisation récurrents (de la même façon que seront présentés les squelettes algorithmiques à la section 1.4). Leur objet est de permettre à une plus large communauté de programmeurs de s'intéresser à la programmation parallèle en proposant une méthodologie au niveau de la conception des algorithmes.

1.3.2.3 Exemples d'outils de développement

Nous pouvons citer quelques exemples d'outils mettant en œuvre une méthodologie de prototypage rapide.

L'environnement de développement *Proteus* a été développé par les universités de Duke et de Caroline du Nord aux Etats-Unis dans le cadre du projet DARPA "Prototyping Technology Project" durant la première moitié des années 90. Il est basé sur le langage de haut-niveau du même nom [MNP+91] [Nyl91] (en ce sens il se rapproche des langages spécialisés évoqués à la section 1.2.3 page 30). Ce langage est un langage impératif structuré. Il a été conçu pour supporter de manière native un ensemble de modèles de parallélisme comme le parallélisme de données par exemple. Des instructions spécifiques permettent d'expliciter le parallélisme à utiliser sans pour autant faire appel à des instructions bas-niveau (comme des instructions de communication explicites). Par exemple, l'utilisateur peut indiquer que plusieurs fonctions sont à exécuter en parallèle mais il n'a pas à s'occuper de la manière dont sont gérées et communiquées les données exploitées par ces fonctions.

Dans la catégorie des approches de type *AAA*, citons l'environnement *SynDEx* (Synchronous Distributed Executive) conçu et développé à l'INRIA de Rocquencourt [LSS91], qui en est le fer de lance. Il se présente sous la forme d'une interface graphique et s'inscrit dans le cadre de l'aide à la parallélisation d'applications de traitement du signal. Il a été développé pour permettre à l'utilisateur d'étudier conjointement l'aspect logiciel et l'aspect matériel du produit final. De ce fait, l'utilisateur doit spécifier sous la forme de deux graphes distincts, l'un pour la partie logicielle l'autre pour la partie matérielle, l'organisation de son application. Côté logiciel, l'algorithme est représenté par un Graphe Flot de Données Conditionné (GFDC). Ce type de graphe induit un ordre partiel sur les opérations permettant ainsi d'exprimer aisément le parallélisme potentiel de l'application. Côté matériel, l'architecture est représentée par un graphe non orienté dans lequel chaque sommet est un processeur et les arcs symbolisent une liaison physique de communication entre deux processeurs. La figure 1.4 donne l'exemple des graphes qu'il est nécessaires de définir sous *SynDEx* (logiciel et matériel) pour représenter une application de calcul d'histogramme sur 8 processeurs. Une fois l'aspect logiciel et l'aspect matériel fournis par l'utilisateur, *SynDEx* réalise l'adéquation des deux [Sor94] [Sor96]. Cela consiste à réduire le parallélisme potentiel de la partie logicielle au parallélisme effectif de la partie matérielle. Cette réduction est obtenue par la transformation du graphe logiciel afin de le mettre en correspondance avec le graphe matériel (voir exemple sur la figure 1.5). Le problème de la mise en correspondance de tels graphes étant connu pour être NP-complet, des heuristiques de placement/ordonnancement sont utilisées. L'ensemble des opérations de placement et d'ordonnancement des processus et des communications est réalisé au moment de la compilation rendant l'exécutif final complètement statique.

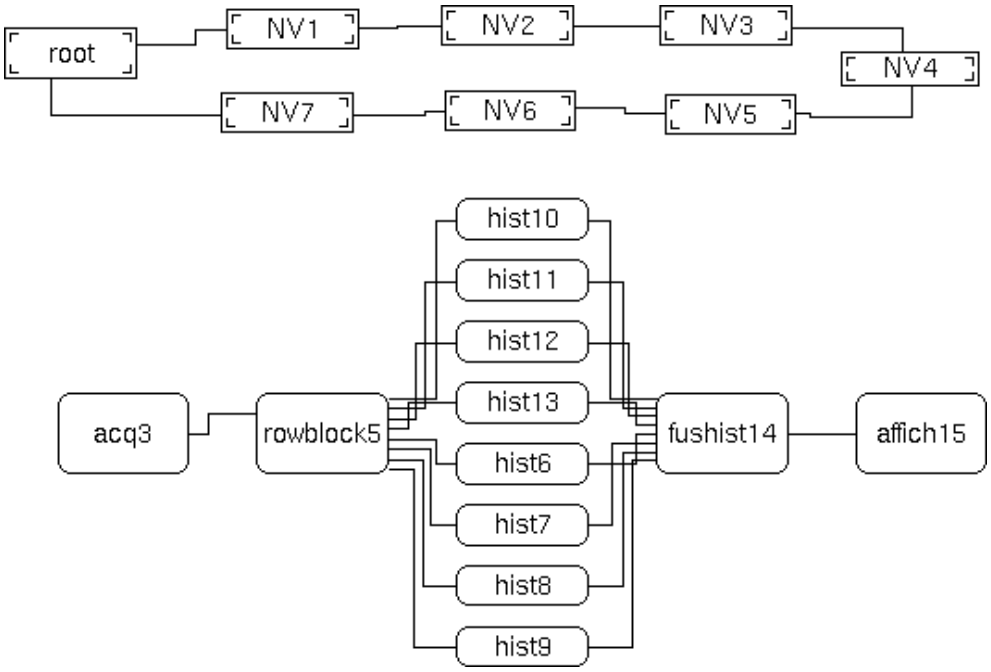


FIG. 1.4 – *Exemple d'application (calcul d'histogramme) spécifiée sous SynDEx.*

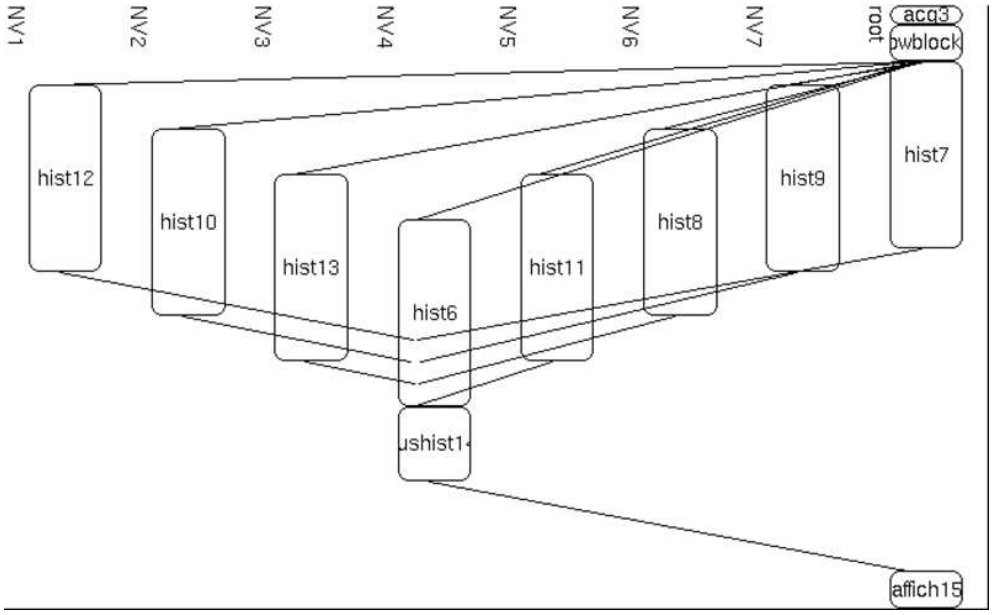


FIG. 1.5 – *Exemple d'adéquation logiciel/matériel calculée par SynDEx (calcul d'histogramme).*

Parmi les approches de type *Design Patterns* se trouve le projet *Enterprise*, qui est issu de recherches menées au département d'informatique de l'université d'Alberta au Canada depuis le début des années 90 [SSLP93] [WSS93] [MSSB00]. L'environnement s'adresse aux réseaux de stations UNIX (implémentation sur Sun). L'utilisateur programme dans cet environnement en utilisant le langage C++, mais dispose également d'une interface graphique spécifique qui lui permet de décrire les aspects parallèles de son programme en termes de sous-divisions hierarchiques dont les noms sont empreintés aux vocabulaires des entreprises : département, service, division, ou travailleur. La spécification du parallélisme par l'intermédiaire de l'interface graphique et la fourniture des fonctions de calcul rédigées en C++ permet à l'environnement de générer le code final de l'application en intégrant les fonctions de l'utilisateur au sein de harnais de communications déduits de la spécification graphique. Les communications entre les différentes fonctions se font sur le mode du passage de messages. Pour le développement de ses programme dans cet environnement, l'utilisateur dispose d'un outil de débogage et d'analyse de performance.

L'université de Floride aux Etats-Unis a elle aussi développé un projet autour de la notion de *Design Patterns* parallèles [MMS99b] [MMS99a] [MMS00]. Le groupe de travail propose un *catalogue de motifs* (*Pattern Language*) pour la programmation d'applications parallèles. Ce catalogue n'est pas simplement une collection de motifs parallèles, mais intègre aussi une méthodologie de conception des applications et des éléments d'expertise pour l'aide au développement (connaissance experte du domaine). De plus, le catalogue est structuré selon trois niveaux d'accès au parallélisme :

1. le haut-niveau aide le programmeur à identifier le parallélisme dans son application (partition du programme en tâches concurrentes) et donc à montrer comment la résolution du problème traité va pouvoir tirer profit du parallélisme,

2. le moyen-niveau lui permet ensuite de tirer profit de ce parallélisme (écriture proprement dite d'algorithmes parallèles),

3. enfin le bas-niveau est une collection de «motifs» dédiés à la mise en œuvre terminale du parallélisme (coordination des tâches parallèles du manière générale).

Conformément à cela, le catalogue se scinde en 4 catégories de motifs (*design spaces*) :

1. FindingConcurrency (haut-niveau),

2. AlgorithmStructure (moyen-niveau),

3. SupportingStructures (niveau «intermédiaire»),

4. ImplementationMechanisms (bas-niveau, permettant la génération du code cible).

Avec une telle structure hiérarchique, le programmeur peut paralléliser son application en tirant profit du niveau le plus adéquat par rapport à son expérience propre. Le plus novice entrera dans le cycle de conception par la catégorie *FindingConcurrency* (où il passera le plus de temps afin d'analyser son problème), catégorie qui le guidera ensuite à travers les trois autres. Un programmeur plus expérimenté pourra, quant à lui, choisir de ne pas utiliser le haut-niveau pour commencer directement par une catégorie de niveau plus faible. Les motifs intégrés dans les différentes catégories présentées ci-dessus ont été définis pour répondre à la plupart des

schémas de parallélisation. Mais comme tout système fondé sur un ensemble de constructeurs, il peut se présenter des cas où les constructeurs proposés ne suffisent pas. Dans ce cas, les auteurs [MMS99a] proposent au programmeur de les aider à compléter le catalogue pour mettre à profit cette nouvelle expérience si cela est possible. Avec la catégorie *FindingConcurrency* le programmeur travaille à faire émerger le parallélisme potentiel de son application en considérant la façon dont il aborde la résolution de son problème. La figure 1.6 montre les motifs inclus dans cette catégorie. En commençant par le motif *GettingStarted*, les flèches indiquent le cheminement du programmeur pour définir le parallélisme potentiel de son application. Les doubles-flèches indiquent une possible analyse en cycle, le programmeur ayant besoin de passer d'un motif à l'autre avant d'obtenir la décomposition finale.

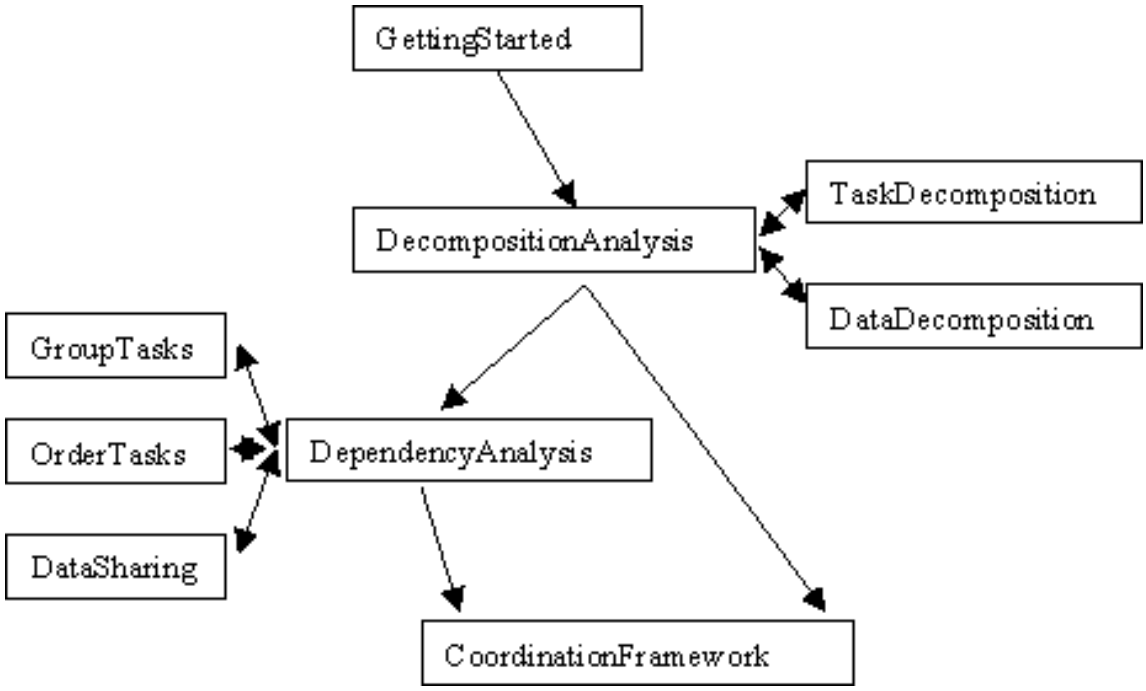


FIG. 1.6 – *La catégorie de motifs FindingConcurrency (extrait de [MMS99a]).*

La catégorie *AlgorithmStructure* s'occupe de structurer l'algorithme en tâches concurrentes. Ici se pose la question de comment exploiter le parallélisme extrait au niveau supérieur. Les motifs de cette catégorie contiennent les stratégies d'exploitation du parallélisme. La figure 1.7 montre les motifs inclus dans cette catégorie. Elle montre l'arbre de décision proposé au programmeur pour l'identification des tâches de son application. Les motifs encadrés d'un double trait fin représentent les motifs intermédiaires. Ils ont pour rôle d'aider le programmeur à naviguer dans le processus de développement à ce niveau. Les motifs encadrés d'un simple trait fin représentent les motifs de décision. Enfin ceux encadrés d'un trait gras sont les motifs terminaux. Ce sont les motifs qui seront effectivement utilisés dans la version parallèle de l'algorithme.

La catégorie *SupportingStructures* peut être vue comme un niveau intermédiaire entre le moyen et le bas-niveau. Les motifs de cette catégorie sont optionnels dans le sens où on peut passer du moyen au bas-niveau sans y recourir forcément. Elle contient essentiellement deux groupes de motifs.

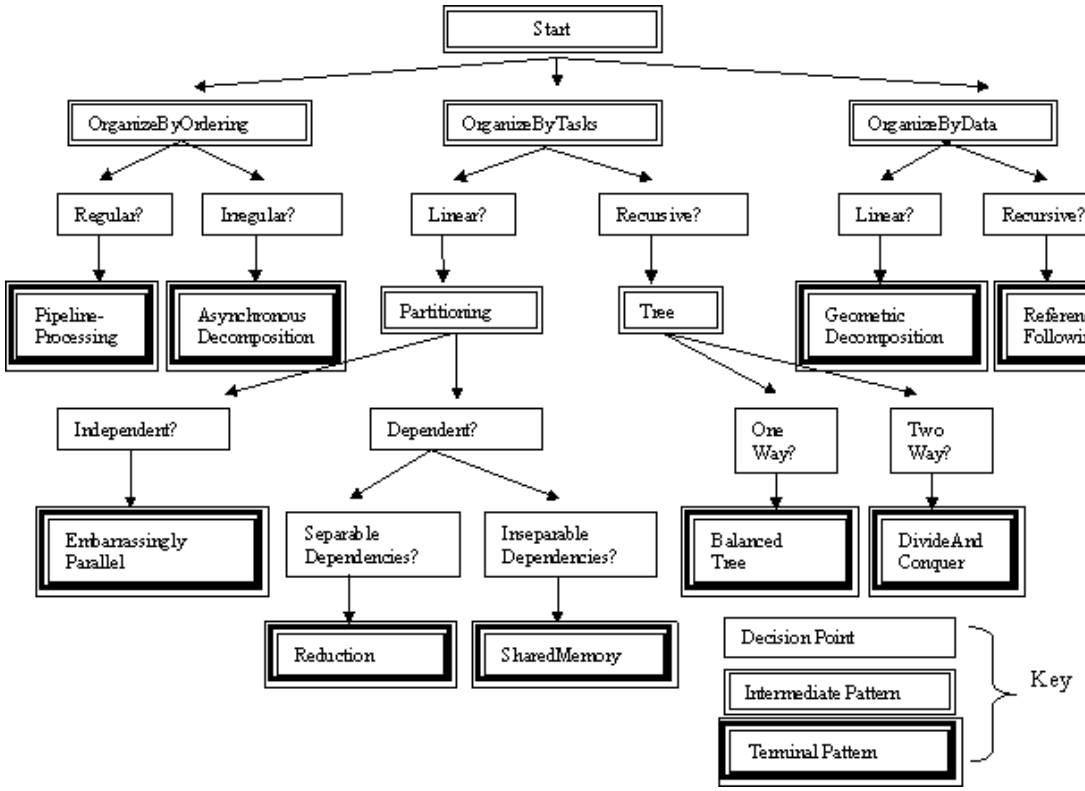


FIG. 1.7 – *La catégorie de motifs AlgorithmStructure (extrait de [MMS99a]).*

Le premier est un ensemble de contructeurs pour structurer les programmes :

- SPMD,
- ForkJoin (un processus démarre plusieurs processus qui réalisent chacun une partie du travail avant de terminer leur exécution sur une synchronisation commune),
- MasterWorker (ferme de processus).

Le second est un ensemble de structures de données communément employées pour le partage de données :

- SharedQueue (queue de messages),
- SharedCounter,
- DistributedArray.

Enfin la catégorie *ImplementationMechanisms* donne la manière dont les motifs des niveaux supérieurs sont effectivement mis en correspondance avec un environnement ou une architecture cible. Ils proposent des descriptions de mécanismes types de gestion des processus, de synchronisation et de communication :

- création et destruction de processus,
- sémaphores, barrières, sections critiques,
- accès asynchone à des ressources partagées,
- opérations de communication élémentaires, groupées.

A proprement parler, les motifs contenus dans cette catégorie (comme ceux de la catégorie précédente) n'en sont pas vraiment (ex. : un mécanisme de communication seul). Les auteurs les ont attaché au catalogue de motifs comme des motifs afin de constituer un «chemin» complet allant du problème au programme final compilé pour une architecture cible. Ainsi même les opérations parallèles de bas-niveau recoivent dans cet environnemnt le même traitement que les méthodes de parallélisation, notamment une documentation structurée et des consignes d'utilisation.

Un autre exemple de l'approche *Design Patterns* est le projet PASM.

B. Preiss et coll. ont développé un modèle générique (indépendant de l'application et des motifs) pour réaliser et utiliser des *Design Patterns* parallèles [GSP99] [GSP00]. Ce modèle est appelé PASM pour *Parallel Architectural Skeleton Model* [GSP01]. Ils définissent ainsi un «squelette architectural», comme la structure représentant tous les attributs d'un motif qui sont indépendants de l'application. Ce modèle est supposé fournir plusieurs fonctionnalités de bibliothèques de communications comme MPI (sur lequel il est implanté), tout en ayant les bénéfices des méthodes associées aux *Design Patterns*. Par conséquent le modèle est bien adapté pour les méthodes classiques de conception orientées-objet et leurs techniques d'implantation. Ainsi le programmeur peut appliquer la méthode de conception logiciel orientée-objet qu'il connait la mieux dans le cadre de la programmation parallèle de son algorithme. PASM se présente comme une bibliothèque de *templates* C++ [GSP98], et de ce fait ne nécessite aucune extension du langage support. Son principal objet est de permettre une extensibilité du nombre de motifs parallèle aisée [GSP99].

Plus précisemment, un squelette architectural est défini comme une collection d'attributs qui encapsulent la structure et le comportement d'un *Design Pattern* parallèle, de manière indépendante de l'application. Le programmeur instancie un «squelette» en complétant les paramètres qui concernent l'application. Cette particularisation donne lieu à une (ou plusieurs) «machines virtuelles» (cf. la figure 1.8 qui schématise le modèle PASM). Chaque machine correspond à une structure particulière de l'application et aux mécanismes de communication et de synchronisation nécessaires. Elle a encore à être complètée par le code proprement dit de l'application pour finalement donner un «module de calcul parallèle».

Les différents motifs proposés, ou définis par le programmeur, peuvent intéragir ou être composés de manière hiérarchique grâce à une interface qui leur est commune [GSP01].

On notera que le modèle développé autorise le programmeur à mixer dans son code l'utilisation des «squelettes» et des primitives de communication bas-niveau (MPI). Le type de machines ciblé par ce modèle est MIMD-DM, et notamment des grappes de PC ou de stations de travail. Dans [GSP99] et [SSS98], on trouve des exemples de programmes conformes au modèle PASM. D'après les auteurs, des comparaisons effectuées entre les mêmes programmes écrits à la main avec des primitives MPI et ceux écrits en utilisant le modèle PASM montrent des différences de performance de seulement $\pm 5\,\%$. Ce chiffre relativement faible, et qui montre une bonne efficacité de l'implantation du modèle, peut se justifier selon eux par la faible «surcouche» que nécessite la réalisation du modèle au-dessus de MPI. Enfin notons que le modèle d'exécution parallèle exploité ici est un modèle SPMD [2]. Tous les processeurs exécutent le même programme, ce qui facilite la gestion du code source (au détriment de l'utilisation de l'espace mémoire).

2. SPMD : Single Program Multiple Data.

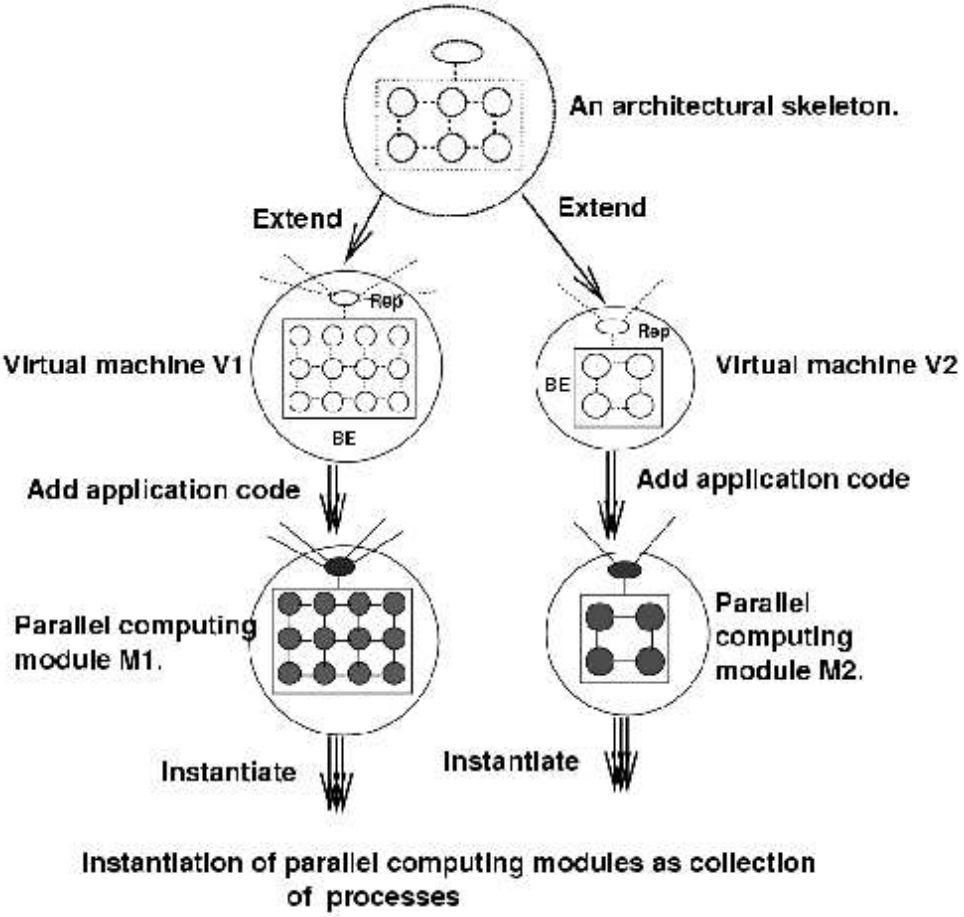


FIG. 1.8 – *Modèle PASM (extrait de [GSP99]).*

1.4 Les squelettes algorithmiques

1.4.1 Définition

Les premières formalisations du concept de squelette algorithmique sont dues à M. Cole [Col89] et D. Skillicorn [Ski90]. Comme le rappelle M. Cole [Col99], la mise en œuvre du parallélisme dans les applications se fait très fréquemment en utilisant un nombre restreint de *schémas* récurrents. Ces schémas explicitent les calculs menés en parallèle et les interactions entre ces calculs. La notion de squelette correspond alors à tout ce qui, dans ces schémas, permet de controler les activités de calcul. Les squelettes algorithmiques sont donc des structures de programmation paramétrables qui encapsulent des schémas de parallélisation communément employés.

L'émergence des squelettes algorithmiques rend compte d'une volonté de *structurer* la programmation parallèle comme la programmation séquentielle a pu l'être. En limitant l'expression du parallélisme à des formes clairement identifiées, ils permettent de gérer la complexité inhérente à la programmation parallèle. Cependant l'identification de telles structures n'est pas chose facile dans la mesure où il est difficile de séparer les différents niveaux de représentation des programmes, ce qui oblige à leur rattacher un grand nombre d'activités : décompostion en processus, placement, ordonnancement, gestion des communications, synchronisations. Par suite cela complique leur définition et leur formalisation, ainsi même que leur utilisation.

Un squelette algorithmique prend donc en charge un schéma de parallélisation précis. Ainsi toutes les opérations bas-niveau nécessaires à la mise en œuvre de la forme de parallélisme choisie sont contenues dans le code du squelette. L'utilisateur paramètre ce squelette en fournissant les fonctions de calcul de son algorithme. Ces fonctions seront alors exécutées en parallèle et les données transiteront entre elles selon le schéma que représente le squelette.

Les environnements de programmation parallèle qui mettent en œuvre une méthodologie fondée sur les squelettes algorithmiques proposent au programmeur un jeu de squelettes (constructeurs génériques paramétrables) correspondant à des formes de parallélisme clairement identifiées et caractérisées dans un domaine applicatif donné. Ce jeu vise à limiter son travail de parallélisation au seul choix des squelettes et de leur composition. Il n'a alors plus qu'à les particulariser avec ses fonctions de calcul.

1.4.2 Atouts

Les atouts communément avancés de cette méthodologie sont les suivants.

Tout d'abord, puisque les squelettes encapsulent tous les détails de la mise en œuvre des formes de parallélisme efficaces utilisables pour un domaine applicatif donné, et par rapport à une architecture cible connue, le programmeur n'a plus à les gérer explicitement. Il est ainsi libéré des tâches fastidieuses de programmation «bas-niveau» et peut d'avantage se concentrer sur les problèmes d'algorithmie, ou évaluer diverses approches de parallélisation de son application. Ce point est un *atout important* en faveur de l'utilisation des squelettes algorithmiques dans le cadre du *prototypage d'applications*, comme nous le verrons plus loin.

Considérant la *fiabilité* de l'implantation d'applications parallèles, puisque la mise en œuvre du schéma de parallélisation des squelettes est effectué une fois pour toutes (pour une architecture cible donnée, voire pour une vaste classe de machines si elle repose sur l'utilisation d'une bibliothèque de communication standard), elle ne peut s'en trouver qu'améliorée. En effet, elle élimine les erreurs provenant de la réalisation de schémas de parallélisation complexes. De même, l'*efficacité* de l'application s'en trouve améliorée puisque la parallélisation proposée dans les squelettes a été produite par des spécialistes du domaine, et non par le programmeur de l'application finale qui ne l'est pas forcément.

Enfin, l'aspect *portabilité* de l'application se trouve améliorée par l'utilisation de squelettes algorithmiques dans la mesure où la parallélisation n'est pas décrite dans l'application en termes d'opérations afférantes à une architecture donnée, mais en termes de schéma *abstrait*.

Soulignons aussi que c'est à travers le choix réduit de squelettes proposés à l'utilisateur que cette approche tente de maîtriser la complexité de la tâche de parallélisation.

1.4.3 Considérations sur les langages pour l'exploitation des squelettes

En pratique, les langages permettant l'utilisation des squelettes algorithmiques (permettant au programmeur de les spécifier pour son application) doivent répondre aux deux exigences suivantes. Premièrement, ils doivent supporter les données et fonctions polymorphes, et deuxièmement, offrir la possiblité de définir des fonctions d'ordre supérieur.

Dans un langage polymorphe, on peut définir des fonctions s'appliquant à n'importe quel type de données en ayant la même définition. Le support des fonctions d'ordre supérieur permet quant à lui d'éviter toute restriction d'usage des fonctions (elles pourront notamment être passées en argument ou retournées comme résultat).

Ces nécessités proviennent de la généricité des squelettes algorithmiques.

C'est une des raisons pour lesquelles les langages fonctionnels tels que ML [Mil84] [MTH90], Caml [INR02] ou Haskell [HPF99a] sont fréquemment employés au service des environnements de programmation fondés sur les squelettes, car le polymorphisme et les fonctions d'ordre supérieur s'y expriment naturellement. Cela étant, l'emploi de tels langages pour spécifier la parallélisation souhaitée d'une application, n'oblige en rien que l'intégralité de l'application soit écrite dans ce langage. Les fonctions de calcul, spécifiques de l'application, peuvent notamment continuer d'être écrites dans un langage impératif «classique» (C et Fortran par exemple).

1.4.4 Eléments de comparaison avec les *Design Patterns*

Même si les squelettes algorithmiques ont été développés de manière complètement indépendante des *Design Patterns*, leur philosophie est très proche de celle qui sous-tend ces derniers.

Malgré cela, les deux approches présentent des différences dont les points essentiels sont leur expressivité, la manière dont ils sont implantés, mais aussi les possibilités d'extensibilité. Concernant l'expressivité des deux approches, M. Danelutto note dans [Dan01] que les squelettes algorithmiques ont plus de potentiel du fait de leur nature fondamentalement déclarative. Ainsi, il est plus facile de distinguer d'un côté l'expression du parallélisme pour un algorithme et de l'autre le code séquentiel de calcul qu'on ne peut le faire avec l'approche *Design Patterns*. Pour ce qui est de leur implantation, l'approche orientée-objet qu'induit classiquement l'utilisation des *Design Patterns* représente un grand intérêt pour l'introduction de ces méthodes de programmation dans la communauté informatique. Qui plus est, un autre avantage de cette approche est la manière relativement simple avec laquelle le portage d'un motif peut être fait. En effet, il suffit de limiter les classes constitutives du motif à des sous-classes limitées aux caractéristiques fonctionnelles de l'architecture parallèle ciblée. Cela étant, un résultat similaire peut être obtenu avec l'approche par squelettes si ces derniers sont développés sur un modèle de communication «en couches» en utilisant des bibliothèques standard de communication comme MPI (voir le chapitre 4). Dans le premier cas, avec les *Design Patterns*, le «sous-classement» rend explicite la limitation de l'implantation à une architecture cible donnée, et peut ainsi, éventuellement, renseigner l'utilisateur sur les possibilités de son système ; alors que dans l'autre cas l'opération est totalement transparente, ne demandant aucune modification de tout ou partie des caractéristiques opérationnelles des squelettes. Enfin, l'utilisation des *Design Patterns* présente un bon potentiel d'extensibilité, c'est-à-dire la possibilité d'ajouter un nouveau motif de parallélisation à la base déjà existante ; c'est même tout l'intérêt du concept. Ici l'accès «protégé» aux classes constitutives d'un motif, du fait de l'utilisation de langages et méthodes orientées-objet pour son développement, garantit que le nouveau motif de parallélisation soit pris en charge par des experts du domaine, et soit complètement certifié et documenté. B. Massingill [MMS99b] note aussi que la différence avec les squelettes algorithmiques tient à ce que l'approche par *Design Patterns* met l'accent sur la réutilisabilité du processus de conception, plus que sur celle du code. De plus, ils sont plus aptes à traiter le processus de conception à différents niveaux (de l'extraction du parallélisme d'une application à son implantation effective).

M. Danelutto dans [Dan01] fait une synthèse des éléments constitutifs de l'approche *Design Patterns* parallèle et les mets en concurrence avec les squelettes algorithmiques. Nous reprenons dans la table 1.1 les arguments essentiels de cette comparaison.

	Design Patterns	**Squelettes algorithmiques**
Type de programmation	Parallèle et distribuée	Parallèle
Niveau d'abstraction	Elevé	Très élevé
Imbrication	Possible	Possible
Réutilisation du code	Possible (et encouragée !)	Possible (et encouragée !)
Langage support	Orienté-objet de préférence	Tout langage
Extensibilité	Très élevée	Limitée

TAB. 1.1 – *Comparaison des approches Design Patterns et Squelettes algorithmiques (d'après [Dan01].*

1.4.5 De l'intérêt des squelettes algorithmiques pour le prototypage rapide

L'utilisation de structures comme les squelettes algorithmiques facilite la mise en œuvre du prototypage rapide en programmation parallèle. En effet, elles correspondent à l'une des nécessités que nous avons déjà évoquées plus haut, à savoir la séparation de la spécification du parallélisme et de la méthode de calcul, dans le sens où une fonction de calcul ne contient aucune référence à une quelconque opération relative à la mise en œuvre d'un schéma de communication. Si les fonctions de calculs ont bien à être développées en tenant compte du schéma de parallélisation dans lequel elles vont venir s'insérer, les éléments opérationnels de ce schéma (instructions de communication, synchronisations,...) n'y sont pas inscrits. Les deux spécifications peuvent alors être manipulées conjointement mais séparément ; ceci accroît les possibilités d'évaluation de plusieurs solutions pour un problème donné.

La mise en œuvre effective du parallélisme (instructions de communication, synchronisations,...) est, par l'intermédiaire des squelettes de parallélisation, complétement cachée à l'utilisateur. Ce dernier ne se concentre que sur la forme de parallélisme qu'il souhaite employer, mais n'opère jamais aucune action pour l'obtenir si ce n'est de choisir le(s) squelette(s) le(s) plus adéquat(s) pour son algorithme. (La question du «quoi» l'emporte sur celle du «comment».)

Il est clair qu'une telle méthode facilite le travail de l'outil de parallélisation puisque le code correspondant à la mise en œuvre du parallélisme et celui correspondant à la technique de calcul (code de l'utilisateur) sont deux entités séparées qui n'interagissent qu'au moment de l'exécution. De plus, même à ce niveau-là, l'utilisation des squelettes de parallélisation reste une méthode *non-intrusive* puisque les deux codes peuvent rester séparés dans des unités de compilation distinctes (aucune instruction spécifique au parallélisme, et donc à l'architecture sous-jacente, n'est présente dans le code des fonctions de calcul de l'utilisateur). On évite ainsi que le parallélisme apparaisse dans l'écriture de l'algorithme de l'utilisateur. De ce fait, et dans l'idéal, le squelette utilisé pour la parallélisation d'un algorithme (et donc le schéma de parallélisation sous-jacent) peut facilement et rapidement être remplacé par un autre pour évaluer une nouvelle combinaison entre ceux-ci et l'algorithme de l'utilisateur [3]. Cette technique permet donc de séparer l'aspect purement calculatoire d'un algorithme de la manière dont il est parallélisé. La séparation de ces deux spécifications est essentielle pour ne pas entrelacer dans le même code «instructions de calcul» et «instructions de communication».

Qui plus est, les schémas de parallélisation étant encapsulés dans les squelettes algorithmiques, cette technique tend à réduire les erreurs de programmation dues au parallélisme lui-

3. En pratique, le changement de squelette peut requ´erir la modifi cation de l'interface des fonctions s´equentielles, voire dans une certaine mesure les op´erations qu'elles effectuent.

même. Ce dernier n'apparait plus directement dans l'algorithme de calcul. Il représente la façon dont sont liées et communiquent ensembles les différentes fonctions de calcul. En réduisant le temps consacré à la recherche d'erreurs induites par la programmation de la parallélisation, on peut ainsi réduire le temps de développement de l'application. (Ici, la «sélection» de solutions existantes est privilégiée par rapport à la «réinvention» de solutions plus ou moins nouvelles.)

Enfin, du fait même qu'il n'est pas nécessaire d'introduire des instructions concernant la parallélisation dans le corps des algorithmes de calcul, cette méthode augmente la capacité de réutilisation de codes séquentiels déjà existant en vue de leur parallélisation. On peut ainsi potentiellement prototyper plus rapidement une application séquentielle sous forme parallèle pour évaluer le gain que cela peut procurer que si toute l'application devait être réécrite. Bien entendu, cela ne signifie nullement que l'utilisateur ne doive pas apporter quelques modifications au code d'origine. En effet, il est toujours nécessaire de l'adapter aux contraintes imposées par l'environnement de développement utilisé (interfaçage des fonctions séquentielles, utilisation des variables de portée globale, manipulation de données réservées à l'environnement,...). Il est de plus souvent nécessaire de restructurer l'algorithme d'origine afin de mettre en évidence le parallélisme.

De manière indirecte, l'exploitation des squelettes algorithmiques pour la mise en œuvre d'outils de prototypage rapide facilite la portabilité des applications développées. Les instructions de parallélisation étant regroupées dans le code des squelettes, l'utilisateur ne pourra pas insérer dans son propre code des instructions qui seraient spécifiques à une machine donnée. Ainsi, en programmant les squelettes de manière à utiliser une bibliothèque standard de communication (MPI, PVM,...), le développeur de l'outil peut garantir que le code exécutable sera portable sur toutes les plate-formes disposant de cette bibliothèque après une simple recompilation (voir figure 1.9).

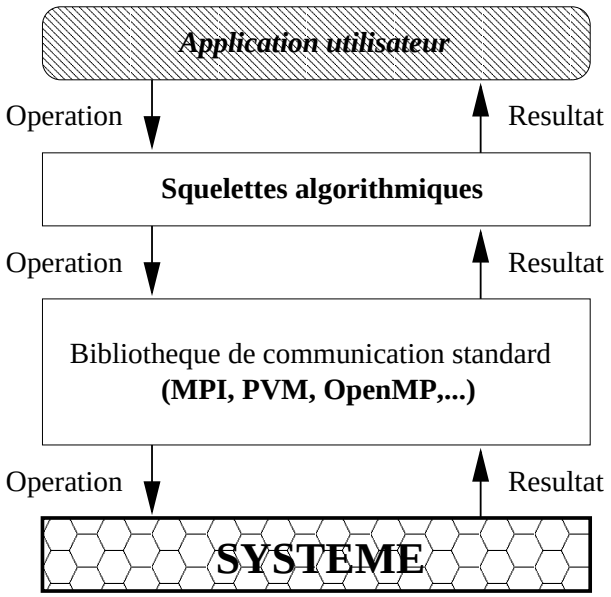


FIG. 1.9 – *Portabilité d'une application exploitant des squelettes écrits avec une bibliothèque de communication standard.*

1.4.6 Caractéristiques d'un outil de prototypage rapide exploitant les squelettes algorithmiques

Ces outils reposent sur le principe commun de demander à l'utilisateur de spécifier le ou les schémas de parallélisation qu'il souhaite pour son algorithme en les sélectionnant parmi un ensemble prédéfini. Cette spécification est plus ou moins explicite en fonction des outils. SKiPPER est quant à lui un environnement complètement explicite : le choix d'un squelette est une indication – impérative – de parallélisme. L'environnement développé à l'université d'Heriot-Watt en Ecosse (cf. la section 1.5.3.2) est lui plus implicite dans la mesure où la spécification de certains squelettes (comme le `map`) est une suggestion adressée au compilateur sur la mise en œuvre du parallélisme. Ces schémas sont proposés de manière totalement indépendante de l'application de l'utilisateur. Ils peuvent donc ainsi constituer une bibliothèque (extensible ou non) de schémas de parallélisation sous forme de codes réutilisables à volonté. Ainsi l'utilisateur fourni les fonctions de calcul spécifiques à son application et l'outil de prototypage se sert des schémas de parallélisation qu'il a choisis pour produire automatiquement le code nécessaire à la mise en parallèle de ces fonctions. On retombe alors sur l'aspect «non-intrusif» de la méthode.

Avec cette approche, le développement d'une application parallèle est alors un compromis entre, les performances recherchées d'une part, et le gain en portabilité et en facilité de programmation d'autre part.

1.5 Application à la vision artificielle

1.5.1 Intérêt de l'approche

Le développement de la recherche en vision artificielle a donné lieu a l'émergence d'algorithmes efficaces pour résoudre une classe de problèmes donnée (filtrage, extraction de contours, approximation polygonale, groupements perceptifs, reconnaissance de formes, suivi d'objets, *etc.*). Ces algorithmes sont acceptés comme références en matière de traitement d'images et sont utilisés comme tels. Partant de cet état de fait, on comprend aisément l'intérêt d'une approche comme le prototypage rapide et les squelettes algorithmiques appliquée à ce domaine d'applications. En effet, on se retrouve fréquemment face à des problèmes qui se résolvent en associant plusieurs briques de base que sont les algorithmes de traitement d'images standard (même si la recherche dans ce domaine reste très active et que des problèmes nouveaux ou pointus obligent régulièrement au développement d'algorithmes entièrement originaux). Dans ce cas, on peut appliquer des schémas de parallélisation à chacun de ces algorithmes. Inversement, l'extraction de schémas de parallélisation récurrents sur cette base permet de construire des squelettes adaptés à la problématique de la vision artificielle (démarche ascendante pour la constitution de la bibliothèque, et démarche descendante pour son utilisation).

Donc, l'intérêt de l'approche est dans un premier temps qu'elle peut être construite à partir de l'existant, c'est-à-dire que la méthode découle du substrat qu'est le domaine d'application. Les squelettes de parallélisation vont être élaborés à partir de l'expérience acquise dans le domaine, et non pas défini *a priori*.

On peut ainsi distinguer, dans le cadre de traitements d'images de bas et moyen niveaux, cinq grandes catégories de squelettes.

La première est dédiée aux *traitements iconiques*. Ces traitements effectuent des transformations de type image vers image. Ces schémas correspondent à un parallélisme de données fixe dans lequel un même flot d'instructions opère sur chaque pixel de l'image d'entrée. Pour ce faire, l'image est subdivisée en sous-domaines, normalement de même dimension. Chaque sous-domaine doit pouvoir être traité de manière indépendante des autres. Il est alors procédé à une simple concaténation des différents résultats pour former l'image de sortie. Les traitements classiques faisant partie de cette classe sont par exemple les convolutions et autres opérations de filtrage.

La seconde s'applique à l'*extraction d'éléments caractéristiques* dans une image. La première phase du traitement consite à procéder à un découpage de l'image comme précédemment. Les traitements qui sont réalisés ne produisent alors plus une image en sortie, mais un ensemble d'informations caractérisant l'image qui sont fusionnés par une fonction spécifique. Parmi les traitements entrant dans cette catégorie, citons le calcul d'histogrammes et la segmentation au sens contour [Cou96].

La troisième traite des *listes de données*. Ces traitements s'insèrent dans les traitements de moyen-niveau. Les données sont le plus souvents des attributs de l'image, extraits par exemple par des opérateurs appartenant à la catégorie précédente. Le parallélisme mis en œuvre ici reste un parallélisme de données comme pour les deux catégories précédentes. La différence est qu'il est ici variable : la dimension de la liste des données et le temps de traitement de chaque donnée dépendent de l'image à traiter, informations qui ne peuvent généralement pas être prédites (à la compilation). Les schémas habituellement retenus pour paralléliser ces algorithmes sont de type *ferme de processeurs* où un processeur spécialisé distribue les données à traiter à un ensemble de processeurs de traitement exécutant tous le même flot d'instructions. Il est clair que ce schéma est fondamentalement dynamique, c'est-à-dire que la distribution et la collecte des données ne peuvent respecter un ordonnancement prévu à la compilation, mais doivent être ordonnancés à l'exécution en fonction des données elles-mêmes. Cela est nécessaire pour garantir l'équilibre de charge des processeurs de traitement. Les algorithmes entrant dans cette catégorie sont par exemple les algorithmes d'approximation polygonale de contours et d'extraction de groupements perceptifs [Cou97].

La quatrième catégorie fait intervenir des *traitements récursifs*, notamment ceux mettant en œuvre un schéma de type *divide-and-conquer*. En fait cette catégorie d'algorithmes englobe les précédentes mais ici le traitement des données peut générer récursivement d'autres données à traiter elles-aussi. L'exemple type sont les algorithmes de segmentation en région par division-fusion [Cha91].

Enfin la dernière catégorie traduit la nature *itérative* des applications de vision *réactives*. Ces applications ne manipulent pas seulement une seule image, mais un flot continu d'images sur lesquelles elles doivent répéter les mêmes opérations. De plus, le résultat calculé pour une image donné peut dépendre des résultats sur les dernières images traités. C'est par exemple le cas avec les algorithmes de détection de mouvements et de prédiction-vérification (Kalman) [Mar00].

De ces catégories on retiendra notamment l'importance des deux premières fondées sur un parallélisme de données qui correspondent aux traitements de bas-niveau qui se retrouvent dans la quasi-totalité des applications de vision. Ils possèdent donc un fort potentiel de réutilisation, et représentent pour certaines applications jusqu'à 90 % du temps de calcul total de l'application [Cou97].

Dans un second temps, étant donné le fort taux de réutilisation potentiel d'algorithmes de vision se retrouvant impliqués dans de multiples chaînes de traitement d'images (filtres, algorithmes de segmentation,...), les squelettes pouvant servir à leur implantation sont potentiellement adaptés au développement de nouveaux algorithmes. Ces schémas de parallélisation pourront alors soit servir à construire une séquence d'algorithmes de vision déjà connus, où bien à en élaborer de nouveaux en ayant à disposition des schémas *adaptés* au domaine applicatif cible.

Un autre intérêt de l'approche par squelettes est la prise en compte de contraintes temporelles sévères dans les applications de vision artificielle. En effet, dans ce cas, on préférera utiliser des architectures dédiées permettant de les satisfaire potentiellement. Les squelettes permettent alors d'intégrer de manière transparente les caractéristiques cruciales de la machine au schéma de parallélisation. L'utilisateur continue d'utiliser un squelette qu'il connait, mais lors de l'implantation le schéma effectivement utilisé peut être optimisé pour l'architecture cible sans que cela soit apparent pour l'utilisateur.

1.5.2 Restriction fondamentale

L'objection qui peut être avancée quant à l'utilisation des squelettes de parallélisation dans le cadre des applications de vision artificielle est l'incertitude quant à la complétude de la «base» de squelettes[4]. En effet, rien ne permet de prouver que le jeu de squelettes mis à la dispostion de l'utilisateur permet de résoudre tous les problèmes pouvant survenir dans le domaine d'application en matière de parallélisation.

Face à cela, on peut être tenté de proposer une base extensible, c'est-à-dire permettre à l'utilisateur de créer ses propres squelettes s'il note une carence dans les possibilités de spécification de son algorithme avec les squelettes existants. Deux questions se posent alors.

- D'une part cette facilité risque d'engendrer une dérive dans la méthodologie aboutissant à une redondance des squelettes dans la base. En effet, avec cette possibilité, l'utilisateur a le choix entre créer de toute pièce un nouveau squelette ou composer des squelettes existants lorsqu'il est confronté à une difficulté. La création immodérée de nouveaux squelettes peut faire perdre l'avantage de la méthode qui est de proposer un panel restreint de schémas de parallélisation afin de maîtriser la complexité de la tâche de parallélisation.
- D'autre part l'utilisateur risque d'être tenté par le développement de squelettes spécifiques à chaque nouvel algorithme qu'il souhaite concevoir. Là encore la philosophie sous-jacente de l'approche ne serait plus respectée.

Il faut donc veiller à ce que cette possibilité ne soit utilisée que pour venir corriger un défaut de la base de squelettes. A cela il faut encore ajouter le problème technique qu'engendre la mise à disposition d'une base de squelettes ouverte. En effet, un squelette déjà intégré à la base est un squelette dont l'environnement de programmation possède toutes les caractéristiques opérationnelles pour l'exécuter. Lorsque c'est l'utilisateur qui en crée un, il faut lui donner les moyens d'informer l'environnement sur toutes les caractéristiques comportementales et architecturales éventuellement du nouveau squelette pour qu'il intéragisse convenablement avec les autres.

4. En fait, le fait de parler de *base* de squelettes est inapropri´e : aucun r´esultat ne vient garantir que l'ensemble que forme les squelettes s´electionn´es pour notre environnement constitue bel et bien une base au sens math´ematique du terme.

Une autre réponse (plus pragmatique) peut être apportée à l'objection sur la complétude de la «base» de squelettes : «s'assurer», par construction, que la «base» est suffisante en suivant une démarche ascendante (analyse rétrospective d'algorithmes existants). C'est la démarche suivie dès l'origine pour le projet SKiPPER.

1.5.3 Quelques réalisations

Très peu de travaux s'intéressant au prototypage rapide par l'intermédiaire des squelettes algorithmiques ont eu comme champ applicatif le traitement d'images. Parmi ceux-ci, citons les projets de l'université de Pise [Pel93][BCD+97] et de l'université d'Heriot-Watt à Edimbourg [MS95] [SBMK98]. Comme l'outil SKiPPER [Gin99] [SGCD01], décrit au chapitre 2, ils ont la caractéristique commune d'avoir été validés sur des applications de vision artificielle réalistes (reconnaissance d'écriture par exemple pour le projet de l'université de Pise [Pel93], et identification d'objets 3D pour celui de l'université d'Heriot-Watt à Edimbourg [MS95] [SMW96]).

1.5.3.1 P^{3}L

P^{3}L (Pisa Parallel Programming Language) est un environnement de programmation parallèle reposant sur l'emploi du langage de haut-niveau du même nom. Il a été développé conjointement par l'université de Pise et le centre de recherche Hewlett-Packard à Pise en Italie au début des années 90 [Pel93] [BCD+97]. Ce projet est orienté vers le parallélisme massif de type MIMD-DM[5]. Son développement a donné lieu à l'émergence de produits commerciaux comme SKiE [BDPV99] [BCP+98] exploité par la société QSW Ltd.

Les squelettes sont au nombre de sept :

- *sequential* (parallélisme de contrôle),
- *pipe* (parallélisme de tâches),
- *farm* (parallélisme de tâches),
- *loop* (parallélisme de contrôle),
- *geometric* (parallélisme de données),
- *map* (parallélisme de données),
- *reduce* (parallélisme de données).

5. MIMD-DM : Multiple Instruction stream, Multiple Data stream - Distributed Memory (voir [Fly66]). Avec ce type de machine, on dispose d'un r´eseau de processeurs pouvant ex´ecuter leur propre programme ind´ependamment les uns des autres, et disposant de leur propre m´emoire vive. Cette m´emoire se trouve donc localis´ee sur chaque nœud du r´eseau. Les processeurs ext´erieurs au nœud ne peuvent acc´eder au contenu de cette m´emoire directement ; elle n'est pas partag´ee comme dans les machines de type MIMD-SM (Shared Memory) o`u une mˆeme m´emoire physique sert `a tous les processeurs du r´eseau.

Le squelette *sequential* encapsule le code des fonctions séquentielles de calcul de l'agorithme.

Le squelette *pipe* exprime un parallélisme de type pipeline.

Le squelette *farm* représente une ferme de processeurs équivalente à celles définies dans les autres environnements. Un processus maître gère un ensemble de processus esclaves en leur envoyant des données à traiter au fur et à mesure que ces dernières sont disponibles. Il centralise de même les résultats.

Le squelette *loop* permet de tenir compte de la nature itérative de certains algorithmes.

Le squelette *geometric* réalise un parallélisme de type géométrique sur un tableau de données de taille quelconque.

Le squelette *map*, est un cas particulier du squelette précédent : les traitements sont tous faits de manière indépendante les uns des autres. La partition initiale des données est ditribuée sur les processeurs pour traitement.

Le squelette *reduce* réalise une réduction sous forme d'arbre binaire en utilisant un opérateur associatif.

Le langage mis à la disposition du programmeur est un langage de haut-niveau, structuré, qui s'appuie sur le langage C++ [BDO+95]. Le parallélisme y est explicite, c'est-à-dire non-automatiquement déduit du code de l'utilisateur par l'environnement, mais indiqué par l'utilisateur lui-même grâce à un certain nombre de constructeurs spécifiques.

La compilation d'un programme se fait en trois étapes [DOPV94].

La première consiste à obtenir une représentation intermédiaire du code de l'utilisateur sous forme d'un arbre de squelettes. La seconde transforme l'arbre obtenu en un graphe de processus. En effet, chaque squelette est considéré comme un groupe de processus communiquants. Après d'éventuelles optimisations l'arbre de squelettes est donc converti en un graphe de processus destinés à être exploités sur une machine virtuelle appelée P^3M. L'intérêt premier de l'existence de cette machine virtuelle est de repousser dans une troisième phase seulement la génération d'un code cible complètement dépendant d'une architecture physique de machine.

Comme dans les autres projets fondés sur les squelettes, ce groupe de travail part du principe qu'un programme parallèle peut être scindé en deux parties distinctes ; mais ces parties ne font pas référence au découpage classiquement retenu entre fonctions de calcul et schéma de communication. Elles sont davantage, d'une part le code de l'algorithme de calcul, et d'autre part le code nécessaire à son adaptation à une architecture cible donnée. Ces deux entités sont respectivement nommées *true algorithmic code* et *machine dependent code*. La différence se situe essentiellement au niveau de la seconde entité qui n'est plus, comme dans d'autres projets, un schéma de communication seul pour faire circuler les données entre les différentes fonctions de calcul – et assurer ainsi leur inter-dépendances –, mais doit correspondre en plus à l'architecture sous-jacente. Cette notion s'entend aussi de la sorte : P^3L distingue le code dépendant de la machine de celui qui lui est indépendant, alors que les autres approches distinguent le code parallèle du code séquentiel.

Pour l'implantation des squelettes, le compilateur fait appel à des *templates* (comme on peut les définir dans des langages orientés-objet comme le C++). Ces *templates* ciblent la machine virtuelle P^3M. Chacun d'eux intégre un modèle de performance pris en compte à la compilation pour optimiser l'implantation par rapport à la machine cible.

1.5.3.2 Travaux de Michaelson et coll.

Les derniers travaux de cette équipe de chercheurs de l'université Heriot-Watt d'Edimbourg en Ecosse [SMH01] portent sur le développement d'un compilateur parallélisant pour le langage fonctionnel *Standard ML* (SML [Pau91]). Le code produit est du C couplé à la bibliothèque de communication MPI [SBMK98].

Le compilateur identifie dans le code de l'utilisateur un certain nombre de fonctions d'ordre supérieur[6] *susceptibles* de donner lieu a une implantation parallèle efficace. Pour cela, à chaque fonction d'ordre supérieur il associe un schéma de parallélisation sous la forme d'un «harnais de communication». La décision *d'instancier* une fonction d'ordre supérieur sous la forme d'un schéma de parallélisation est prise sur la base des informations collectées par le «profiler» avec l'aide de modèles de performances. Il s'agit donc d'une approche *implicite* dans le sens où les fonctions d'ordre supérieur sont vues comme de simples *indications* de parallélisme potentiel (contrairement à SKiPPER où un squelette conduit toujours à une implantation parallèle). Dans ces travaux, un squelette est une *réalisation* d'une fonction d'ordre supérieur. Le compilateur identifie ainsi les fonctions d'ordre supérieur qui font partie d'un ensemble prédéfini. Cet ensemble de fonctions (*map*, *fold* et *compose*) correspond aux fonctions auxquelles est associé un schéma de parallélisation. Chacune d'entre-elles a une représentation sous forme d'un squelette algorithmique. De ce fait, il lui est associé un harnais de communication (écrit en C et utilisant MPI). Ce harnais représente l'ensemble des communications nécessaires à la mise en œuvre sur la machine cible du schéma de parallélisation. Il est paramétrable, notamment avec le code de calcul des fonctions de l'utilisateur. Des techniques de transformation de programmes sont appliquées au code contenant les fonctions d'ordre supérieur de l'utilisateur afin de réaliser d'éventuelles optimisations. *Le parallélisme est exploité ici de manière statique*, c'est-à-dire décidé au moment de la compilation, contrairement à SKiPPER-II pour lequel l'exploitation est intégralement dynamique.

Les squelettes proposés au programmeur sont donc au nombre de trois :

- `map`, qui exploite une ferme de processeurs pour contrôler le travail de processus de traitement chargés de traiter les données d'une liste qui lui est fournie en argument. Au démarrage chaque processus de traitement reçoit une partie de la liste. La taille de cette sous-liste est déterminée à la compilation en fonction de la topologie du réseau. Tous les éléments ne sont pas forcément distribués dès le début, et donc, dans la suite de l'exécution les éléments restants sont distribués au fur et à mesure que la demande en est faite par les processus de traitement, jusqu'à épuisement des données initiales. Il faut noter que le processus maître chargé de gérer la ferme de processeurs ne réalise par lui-même aucun traitement sur les données ; il ne sert qu'à la distribution des données et à la collecte des résultats [SBMK98].
- `fold`, utilise un schéma sous la forme d'un arbre binaire. Contrairement au squelette précédent, tous les processeurs mis en jeu sont des processeurs de traitement effectif, chaque nœud de l'arbre travaillant sur une portion de la liste de données initiale. Si le nœud n'est pas une feuille de l'arbre binaire, sa sous-liste est scindée en trois sous-éléments dont l'un est conservé localement, les autres étant envoyés aux fils du nœud. Les feuilles de l'arbre se contentent d'effectuer les calculs sans redécomposition [SBMK98].

6. Voir annexe A page 177.

- `compose` applique une fonction de calcul sur le résultat d'une autre (les deux fonctions étant données en argument ainsi que la donnée initiale sur laquelle s'applique la seconde fonction) [MSBK00].

La figure 1.10 montre le synoptique du compilateur SML parallélisant.

Avec ce compilateur, l'utilisateur n'a pas besoin d'avoir de notions de programmation parallèle : le compilateur est capable de l'extraire automatiquement pour lui du code en langage fonctionnel qu'il lui présente. Mais cela n'est fait qu'au niveau des fonctions d'ordre supérieur qui font partie de la bibliothèque de squelettes. Aucun parallélisme ne peut être produit pour les fonctions de calcul séquentielles de l'utilisateur. L'utilisateur n'a ainsi qu'à connaître les fonctions d'ordre supérieur (*map, fold, compose*) qu'il peut utiliser pour paralléliser son programme.

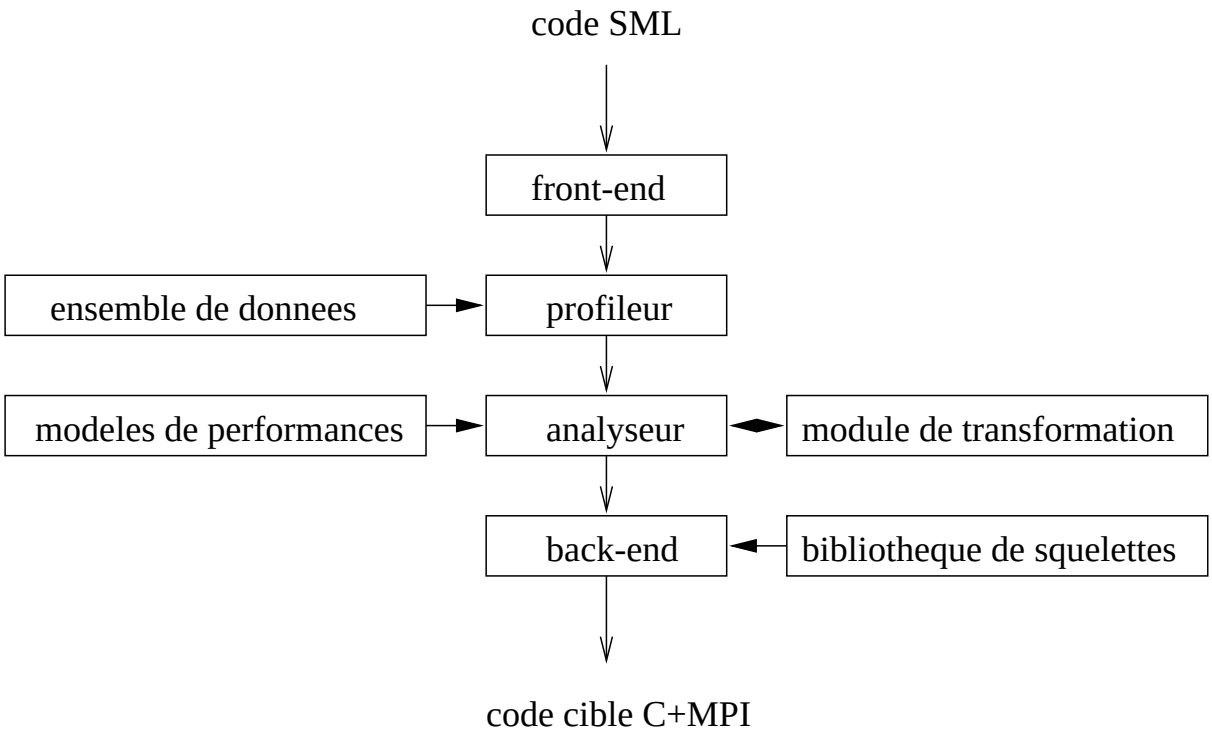


FIG. 1.10 – *Synoptique de la compilation (d'après [SBMK98]).*

Chapitre 2

Le projet SKiPPER : SKiPPER-I

SKiPPER est un environnement d'aide à la programmation parallèle s'appuyant sur l'utilisation des squelettes algorithmiques. Il a été conçu à l'origine pour le prototypage rapide d'applications parallèles dans le domaine de la vision artificielle. Ce chapitre présente les principes sous-jacents à SKiPPER et à sa première version opérationnelle (SKiPPER-I) développée durant la thèse de D. Ginhac [Gin99]. Une revue des limitations de cette première version permettra d'aborder les objectifs et les choix de conception mis en œuvre pour la seconde version de cet environnement, et qui seront détaillés au chapitre 3.

2.1 Présentation et objectifs

SKiPPER (SKeletal Parallel Programming EnviRonment) est un environnement d'aide à la programmation parallèle. Il a été conçu pour permettre le prototypage rapide d'applications de vision artificielle sur machines parallèles. Le machines parallèles visées sont des machines dédiées au traitement d'images et embarquables, de type MIMD-DM [1].

SKiPPER intégre une méthodologie de programmation fondée sur l'utilisation de squelettes algorithmiques. Il propose à l'utilisateur un jeu de squelettes réputé répondre aux problèmes algorithmiques rencontrés en vision artificielle. Ce jeu – limité – de squelettes permet à l'utilisateur de décrire l'organisation, en terme de parallélisme, de son application.

Cette méthodologie a pour objectif de réduire de manière significative le temps global de conception, d'implantation et de validation des algorithmes sur la machine cible [SGCD01]. A travers l'outil SKiPPER, cette méthode de programmation veut donc faciliter la programmation des machines parallèles, en prenant en charge les aspects bas-niveau suivants :

– le découpage de l'application en processus,

– le placement de ces processus sur les processeurs de la machine cible,

– l'ordonnancement des communications inter-processus,

– le routage des communications.

Dans cet environnement, deux langages sont utilisés :

– Caml pour la spécification du parallélisme de l'application,

– C (éventuellement C++) pour l'écriture des fonctions de calcul séquentielles.

De fait, les fonctions de calcul sont écrites de manière complètement indépendante de l'organisation globale de l'application, en accord avec le principe des squelettes algorithmiques. L'utilisateur décrit donc à l'aide du langage Caml uniquement les squelettes utilisés et leur inter-dépendances. Le langage Caml étant un langage fonctionnel, les squelettes s'y expriment de manière «naturelle» sous la forme de fonctions d'ordre supérieur (voir annexe A page 177).

En effet, du point de vue programmation, l'une des caractéristiques essentielles des squelettes algorithmiques est leur généricité, c'est-à-dire leur capacité à être spécialisés par des fonctions *a priori* quelconques. C'est pourquoi les squelettes doivent être polymorphes et apparaître comme des contructions d'ordre supérieur. Par polymorphe on entend la possibilité d'accepter plusieurs types de données différents en conservant pourtant la même déclaration. Par ordre supérieur il faut comprendre la capacité d'une fonction d'accepter une autre fonction comme paramètre. Ces caractéristiques ont justifié en grande partie l'utilisation de langages fonctionnels pour la description des squelettes algorithmiques dans beaucoup d'environnements les utilisants, SKiPPER y compris.

Le langage C a été choisi comme langage de programmation pour les fonctions de calcul car il permet la réutilisation sans trop de difficultés de programmes déjà existants (pour le corps des fonctions de calcul, une attention particulière devant être portée à l'adaptation de leur interface [Gin99] [SGCD01]).

1. Voir note page 48.

2.2 Le jeu de squelettes

2.2.1 Préliminaires

SKiPPER est un environnement qui dispose d'un jeu de squelettes en nombre limité. Ces squelettes ont été choisis pour répondre spécifiquement aux exigences des algorithmes de traitement d'images.

Le choix des squelettes intégrés à SKiPPER s'est appuyé pour l'essentiel sur l'analyse d'un panel de programmes déjà existants. A partir de cette analyse, des schémas récurrents de parallélisme ont été identifés [Cha91] [Can93] [BDHR94] [LCD94a] [LCD94b] [Leg95] [Gin99] [Sér02] .

La bibliothèque de squelettes proposée par SKiPPER n'est pas extensible par l'utilisateur ; les squelettes qui la composent sont supposés convenir, si ce n'est à tous les cas de figure, au moins aux cas les plus couramment rencontrés.

2.2.2 Constituants de la bibliothèque

Le nombre de squelettes retenus pour former la bibliothèque mise à disposition de l'utilisateur est de quatre :

- SCM (Split, Compute and Merge),
- DF (Data Farming),
- TF (Task Farming),
- ITERMEM (ITERation with MEMory).

2.2.2.1 Le squelette SCM (Split, Compute and Merge)

Ce squelette regroupe les schémas de parallélisme dédiés au traitement géométrique des données. Son fonctionnement est le suivant.

La donnée d'entrée (typiquement une image) est tout d'abord divisée en un nombre fixe d'éléments par la fonction utilisateur affectée à la phase «split» du squelette. Chaque élément ainsi obtenu est alors traité de manière totalement indépendante par une fonction utilisateur réalisant la phase «compute» du squelette. L'ensemble des résultats des traitements sont alors ensuite regroupés pour former un résultat définitif de l'application de ce squelette. Ce résultat est une combinaison des résultats intermédiaires. La nature de la combinaison est laissée à la discrétion de l'utilisateur par l'intermédiaire de la fonction assignée à la phase «merge» du squelette.

Ce squelette est dit «statique», c'est-à-dire que son schéma de communication est entièrement connu à la compilation. On notera aussi que, si chaque fonction «compute» doit être placée sur un processeur différent pour obtenir une parallélisation réelle des traitements, la séquentialité dans l'enchaînement des fonctions «split», «compute» et «merge» permet de placer sur un même processeur ces trois fonctions (une seule instance de la fonction «compute» en l'occurence).

Il faut bien noter que, du fait de sa nature même, ce squelette impose :

– qu'une même fonction utilisateur joue le rôle de la phase «compute» (plusieurs fonctions distinctes ne peuvent être utilisées simultanément),

– que les durées des traitements ne soient pas (trop) dépendantes des données afin de garantir que toutes les processeurs finiront leurs traitements au même moment.

Ce dernier point est primordial pour garantir l'efficacité du squelette SCM. Dans le cas contraire des processeurs consommeraient du temps à attendre la fin des traitements sur d'autres processeurs. Le schéma de parallélisation ne serait alors plus adéquat, le déséquilibre de charge devenant trop important.

La figure 2.1 est un synoptique du squelette SCM auquel correspond la *sémantique déclarative* (en Caml) ci-dessous[2]. Nous entendons par *sémantique déclarative*, celle utilisée pour permettre au programmeur de comprendre le rôle d'un squelette en dehors de toute considération architecturale. Elle peut être considérée comme l'interface du squelette.

```
> let scm split compute merge x =
      merge ( map compute ( split x ) )
```

A cette sémantique déclarative correspond la *signature* suivante (qui établit notamment le type des fonctions qui seront passées en arguments du squelette) :

```
# val scm :
     ('a -> 'b list) (* fonction de division   *)
  -> ('b -> 'c)      (* fonction de traitement *)
  -> ('c list -> 'd) (* fonction de fusion     *)
  -> 'a              (* donnee                 *)
  -> 'd              (* resultat               *)
```

La figure 2.2 donne quant à elle un exemple d'exécution (placement et ordonnancement) du squelette SCM sur une architecture a quatre processeurs.

Les algorithmes visés par ce squelette sont donc, dans le domaine du traitement d'images, les algorithmes bas niveau de pré-traitement tels que :

- convolutions,

- filtres,

- histogrammes.

2. Se reporter `a l'annexe A page 177 pour un compl´ement d'information sur la syntaxe Caml de la s´emantique pr´esent´ee.

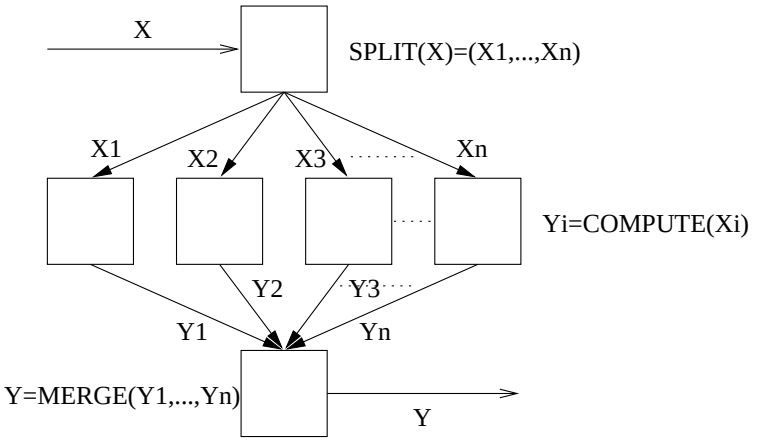


FIG. 2.1 – *Synoptique du squelette SCM.*

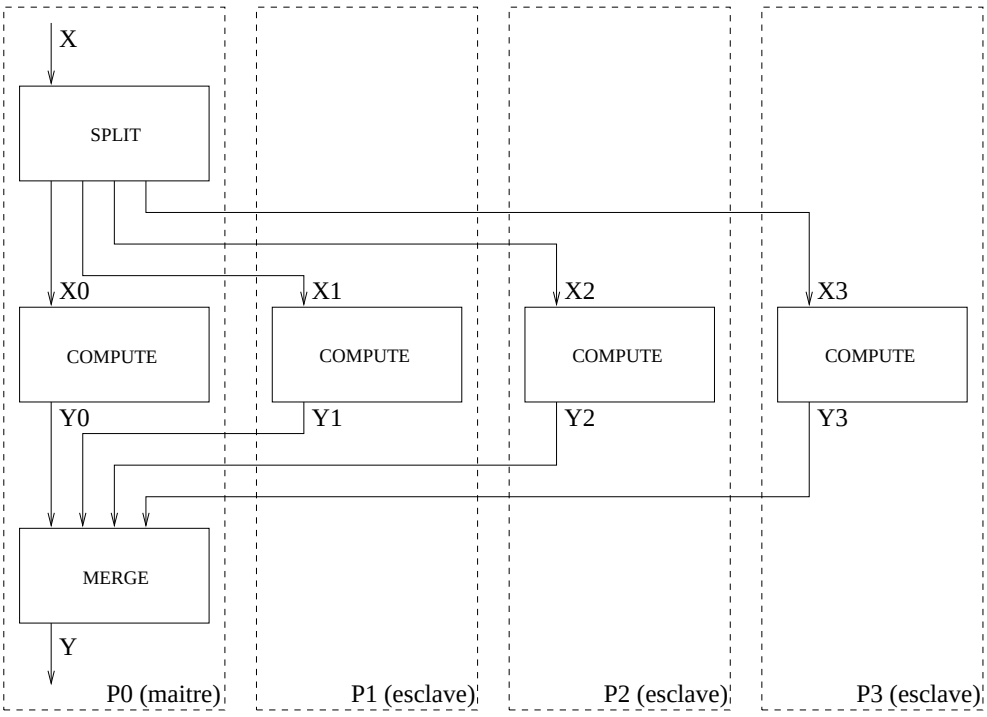


FIG. 2.2 – *Exemple d'exécution sur 4 processeurs du squelette SCM.*

2.2.2.2 Le squelette DF (Data Farming)

Le squelette DF sert à appliquer en parallèle une fonction à tous les éléments d'une liste de données et à combiner les résultats. Il a été spécialement conçu pour gérer les cas où, soit le temps de traitement des données par les processeurs n'est pas uniforme, soit le nombre de données à traiter n'est pas connu à l'avance, voire les deux à la fois. Le temps de calcul dépend alors directement des données qui sont manipulées. C'est notamment le cas lorsque les algorithmes de vision artificielle ne se contentent plus de traiter des images brutes mais des indices visuels (polygônes, segments,...), souvent sous forme de listes d'objets [Can93] [Cou97]. Un mécanisme d'équilibrage de la charge de calcul des processeurs est alors nécessaire. Le squelette DF répartit *dynamiquement* les données d'entrée sur les processeurs pour traitement, tout en réalimentant les processeurs qui auraient fini leur traitement avant les autres.

On notera ici la nature complètement dynamique de ce squelette, le temps de traitement d'une donnée élémentaire sur chaque processeur ne pouvant être anticipé et pouvant être différent de celui obtenu sur d'autres processeurs engagés dans le même schéma de parallélisation.

En résumé, la principale différence entre les squelettes SCM et DF est que le premier encapsule un parallélisme de données *fixe* alors que pour le second il s'agit d'un parallélisme de données *variable*.

Ce schéma utilise pour son implantation un modèle d'exécution en ferme de processeurs.

Un processeur est désigné comme maître, les autres comme esclaves. Le maître est chargé de distribuer les données à traiter aux esclaves. Généralement le nombre de données initiales destinées au traitement étant supérieur au nombre d'unités de calcul disponibles sur la machine cible, le maître garde en réserve un certain nombre de données lorsque tous les esclaves ont été servis. Dès que l'un d'entre eux signale qu'il a terminé son traitement en renvoyant le résultat au maître, ce dernier réalimente l'esclave avec une nouvelle donnée pour le maintenir constamment en charge et ainsi réaliser dynamiquement l'équilibre de charge du réseau de processeurs.

Les résultats en provenance des esclaves s'accumulent au niveau du maître. La manière dont est effectuée l'accumulation est dictée par une fonction utilisateur.

On pourra remarquer que l'ordre d'arrivée des résultats n'est pas forcément celui dans lequel les données ont été distribuées. Enfin, la fonction de calcul opérant pour chaque esclave doit être la même. Ce squelette, comme le SCM, n'autorise pas des fonctions de calcul différentes pour chaque esclave.

La figure 2.3 est un synoptique du squelette DF auquel correspond la sémantique déclarative (en Caml) suivante [3] :

```
> let df compute acc z xs =
      foldl acc z ( map compute xs )
```

3. Se reporter `a l'annexe A page 177.

La signature de ce squelette est :

```
# val df :
    ('a -> 'b)       (* fonction de traitement  *)
  -> ('c -> 'b -> 'c) (* fonction d'accumulation *)
  -> 'c               (* valeur initiale        *)
                      (*   de l'accumulateur    *)
  -> 'a list          (* liste de donnees       *)
  -> 'c               (* resultat               *)
```

La fonction *compute*, comme dans les autres squelettes, est une fonction fournie par l'utilisateur pour traiter les données individuelles. La fonction *acc*, aussi fournie par l'utilisateur, permet d'accumuler les résultats partiels en prevenance des traitements sur les différents processeurs esclaves (*z* est la valeur initiale de l'accumulateur). L'accumulation des résultats partiels est réalisée à mesure que ceux-ci sont produits d'où l'emploi de la fonctionnelle $foldl$ qui permet d'appliquer itérativement la fonction *acc* à la liste de résultats (voire l'annexe A page 177 pour l'expression de $foldl$).

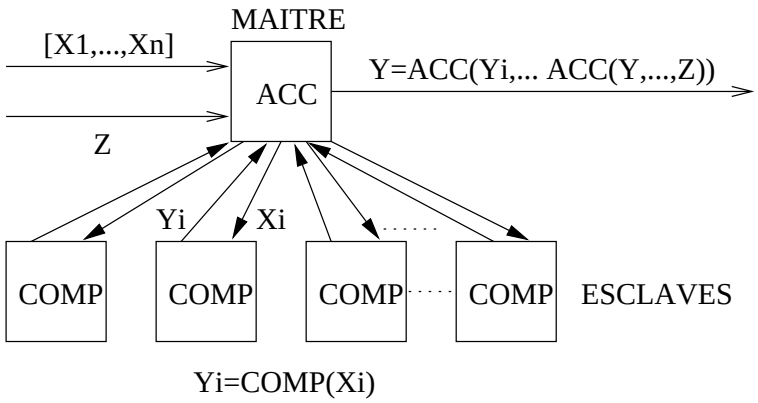


FIG. 2.3 – *Synoptique du squelette DF.*

La figure 2.4 donne quant à elle un exemple d'exécution du squelette DF sur une architecture formée de quatre processeurs. Elle représente le traitement par un squelette DF d'une liste de données à 6 éléments (X0 à X5). Puisque 4 processeurs seulement sont disponibles, seuls 3 sont utilisés comme esclaves et donc réalisent le traitement des données. Le maître commence par envoyer les 3 premières données X0, X1 et X2 à traiter (autant que d'esclaves libres) aux 3 processeurs dédiés aux calculs, pour ensuite se mettre en attente du résultat des traitements sur ces données. La première valeur retournée est Y1 en provenance du deuxième esclave. A ce moment le maître peut envoyer une nouvelle donnée vers cet esclave devenu libre et accumuler ce résultat avant de se remettre en attente à nouveau. Ce processus se renouvelle jusqu'à ce que toutes les données initiales soient traitées.

Les algorithmes visés par ce squelette sont donc, dans le domaine du traitement d'images, des algorithmes dont la complexité dépend des données, comme par exemple les opérateurs d'approximation polygonale de chaînes de points connexes [GG91] [Leg95] [Cou96]. Ces algorithmes exploitent une stratégie récursive de division de la courbe dont l'arrêt est conditionné par la distance séparant la courbe réelle et les segments qui en donnent une approximation. Le temps de traitement dépend ici de la taille et de la forme de la courbe.

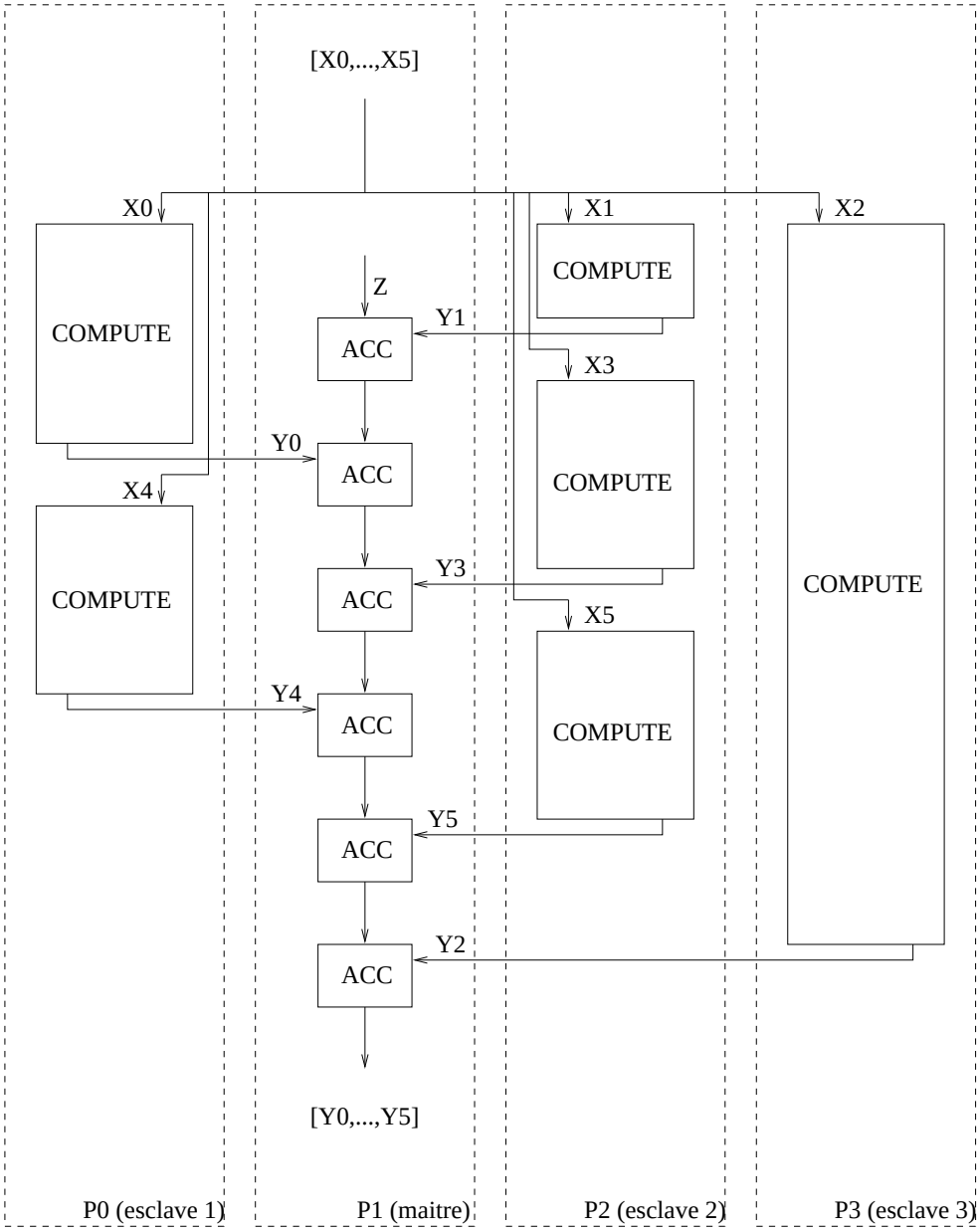


FIG. 2.4 – *Exemple d'exécution sur 4 processeurs du squelette DF.*

2.2.2.3 Le squelette TF (Task Farming)

Le squelette TF est le plus complexe du jeu de squelettes de SKiPPER.

Ce squelette est similaire au squelette DF, et en reprend d'ailleurs les principales caractéristiques. La seule différence de comportement qu'il introduit, et qui le caractérise, est le fait que le résultat d'un traitement peut être, éventuellement, réinjecté comme donnée d'entrée pour subir un nouveau traitement après redécoupage en données plus élémentaires.

En fait, ce squelette peut être considéré comme un squelette DF généralisé en cela que le traitement d'une donnée peut éventuellement générer récursivement de nouvelles données qui seront distribuées à l'itération suivante. Comme pour le squelette DF, c'est une ferme de processeurs qui est utilisée comme modèle d'exécution.

Le maître du squelette TF a pour rôle de distribuer les données à traiter et de collecter les résultats correspondants tout en maintenant l'équilibre de charge en terme de calcul sur l'ensemble de ses esclaves. Cependant, le traitement opéré par les esclaves est un peu plus complexe que l'application d'une simple fonction de calcul à chaque donnée qui se présente. En effet, un esclave commence toujours par appliquer une fonction de prédicat sur la donnée entrante pour savoir s'il doit ou non appliquer la fonction de calcul. F. Chantemargue dans sa thèse [Cha91] donne comme exemple de prédicats les tests d'homogénéité des régions d'une image (tests obtenus par calculs statistiques de moyenne et d'écart-type sur les valeurs valeurs des pixels, une région étant déclarée homogène si l'écart-type sur les valeurs des pixels est inférieur à un seuil fixé au préalable). Un exemple d'utilisation du squelette TF avec ce type de prédicat est donné figure 2.5 [4].

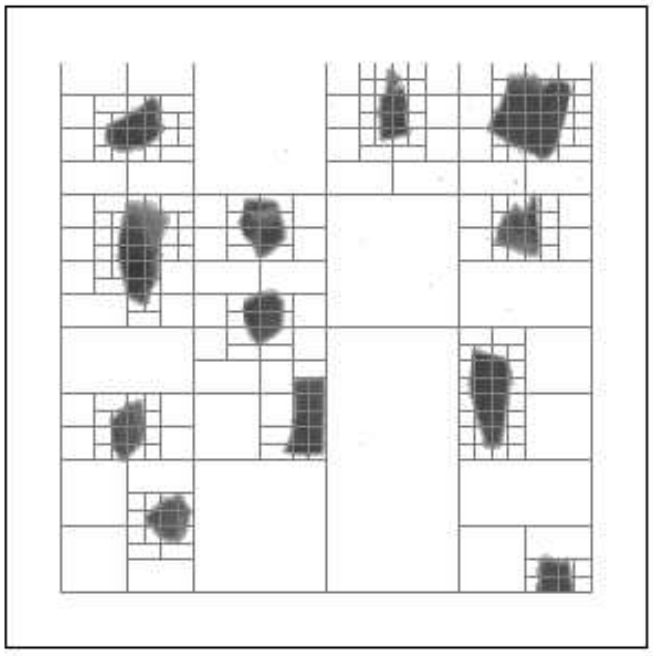


FIG. 2.5 – *Résultat de l'utilisation du squelette TF pour la division récursive d'images.*

4. Se reporter `a la section 5.2.4 page 133 pour une description compl`ete de cet algorithme et l'utilisation du squelette TF pour sa parall´elisation

Si le prédicat est vrai, alors la donnée est traitée localement par l'esclave en lui appliquant la fonction de calcul. Sinon, elle est retournée au maître qui appliquera une fonction de division afin de générer à partir d'elle un nouvel ensemble de données à traiter.

La figure 2.6 est un synoptique du squelette TF auquel correspondent la sémantique déclarative (en Caml)[5] et la signature suivantes :

```
> let rec tf trivial solve divide combine z xs =
      let f x =
          if ( trivial x )
          then
            combine z ( solve x )
          else
            tf trivial solve divide combine z ( divide x ) in
      foldl combine z ( map f xs )

# val tf :
     ('a -> bool)     (* fonction de predicat              *)
  -> ('a -> 'c)       (* fonction de traitement            *)
  -> ('a -> 'a list)  (* fonction de partition             *)
  -> ('b -> 'c -> 'b) (* fonction d'accumulation           *)
  -> 'b               (* valeur initiale de l'accumulateur *)
  -> 'a               (* donnees                           *)
  -> 'b               (* resultat                          *)
```

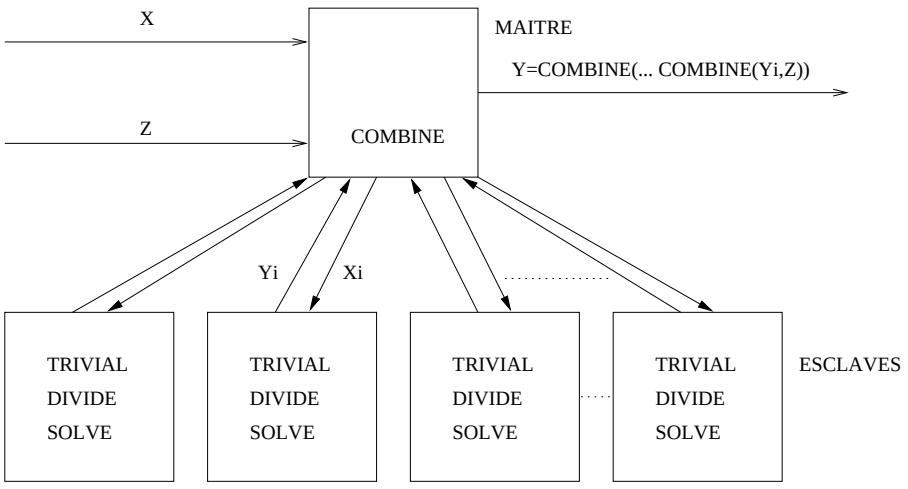


FIG. 2.6 – *Synoptique du squelette TF.*

5. Se reporter `a l'annexe A page 177.

La figure 2.7 donne quant à elle un exemple d'exécution du squelette TF sur une architecture formée de quatre processeurs. Elle représente le traitement par un squelette TF d'une liste de données à 4 éléments (X0 à X3). Puisque 4 processeurs seulement sont disponibles, seuls 3 sont utilisés comme esclaves et donc réalisent le traitement des données. Le maître commence par envoyer les 3 premières données X0, X1 et X2 à traiter (autant que d'esclaves libres) aux 3 processeurs dédiés aux calculs, pour ensuite se mettre en attente du résultat des traitements sur ces données. La première valeur retournée l'est sur le calcul de X1. Mais cette donnée ne répondant pas au prédicat, deux nouvelles données plus élémentaires X11 et X12 sont produites à partir de X1 et réintroduite dans la liste des données à traiter au niveau du maître. Ces données sont traitées lorsque toutes les données déjà présente dans la file d'attente l'ont été (gestion en mode FIFO[6]).

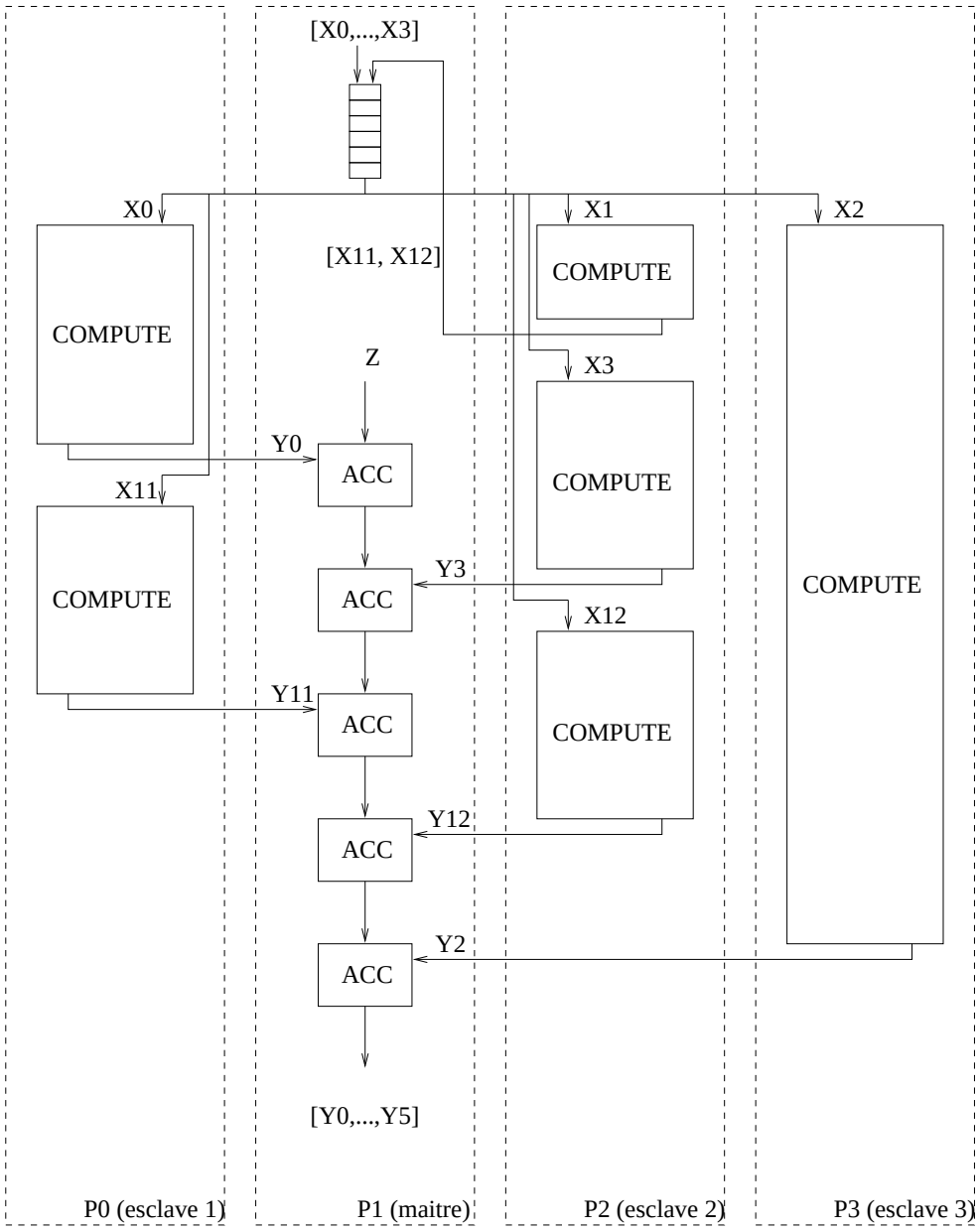


FIG. 2.7 – *Exemple d'exécution sur 4 processeurs du squelette TF.*

6. First In First Out : Premier Entr´e Premier Sorti.

2.2.2.4 Le squelette ITERMEM (ITERation with MEMory)

Le squelette ITERMEM est un squelette de nature particulière. C'est le seul, dans la définition même du jeu de squelettes, à accepter un autre squelette comme paramètre à la place de la fonction de calcul. Son seul rôle est d'itérer une fonction de calcul de l'utilisateur, ou un autre squelette, tout en maintenant en entrée de la fonction, non seulement une donnée à traiter, mais aussi le résultat du traitement précédent. Ainsi la fonction de calcul est capable d'opérer en ayant une mémoire des traitements précédents.

Son rôle n'est donc pas à proprement parler d'encapsuler un schéma de parallélisation, mais de rendre explicite le traitement itératif d'un flux de données, notion essentielle en traitement d'images réactif et temps réel. En effet, dans le cas du traitement d'images issues d'une caméra, c'est un flot d'images, et non une image seule, qu'on a à traiter. Une situation usuellement rencontrée est que le traitement d'une image dépend des traitements effectués sur un certain nombre d'images précédentes (dans le flot video). C'est le cas notamment pour le suivi d'objets en mouvement dans une scène («tracking») [Mar00].

La figure 2.8 est un synoptique du squelette ITERMEM auquel correspondent la sémantique déclarative (en Caml)[7] et la signature suivantes :

```
> let itermem f1 f2 f3 z x =
      let rec h x =
          let z', y = f2 ( z, f1 x ) in
          f3 y; h z' in
        h z
# val itermem :
     ('a -> 'b)           (* fonction d'acquisition        *)
  -> ('c * 'b -> 'c * 'd) (* fonction de traitement        *)
  -> ('d -> unit)         (* fonction de restitution       *)
  -> 'c                   (* valeur initiale de la memoire *)
  -> 'a                   (* donnee d'entree               *)
```

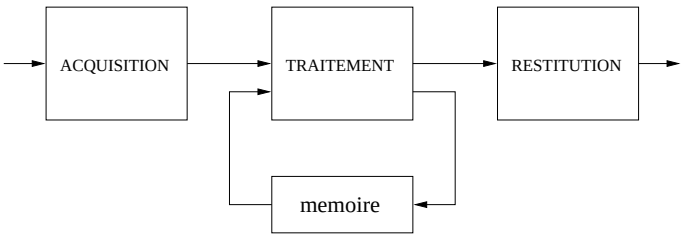


FIG. 2.8 – *Synoptique du squelette ITERMEM.*

7. Se reporter `a l'annexe A page 177.

2.3 L'environnement de développement SKiPPER-I

2.3.1 Présentation

SKiPPER-I est la première version opérationnelle de l'environnement SKiPPER [Gin99]. Comme nous le verrons, elle est basée sur un modèle d'implantation essentiellement statique [8] du jeu de squelettes décrit à la section précédente.

Elle comporte trois étapes de traduction pour faire passer du programme source de l'utilisateur au programme exécutable sur une architecture cible donnée :

1. une étape d'expansion des squelettes,
2. une étape de placement et d'ordonnancement,
3. une étape de génération du code cible.

La première étape prend en charge le code source écrit en Caml pour le traduire sous une forme exploitable par les outils de placement et d'ordonnencement de la seconde phase. On passe ici d'une représentation fonctionnelle à une représentation sous forme d'un graphe flot de données.

La seconde étape exploite ce graphe pour réaliser le placement et l'ordonnancement de ces processus sur une architecture parallèle cible choisie par l'utilisateur. Pour ce faire, l'outil SynDEx de l'INRIA est utilisé [GLS99]. Le code produit lors de cette phase est un macro-code indépendant le langage cible et du processeur utilisé par les nœuds de la machine parallèle cible, mais déjà dépendant de l'architecture et du réseau de communication utilisé *in fine*.

La troisième étape enfin traduit le macro-code en langage C. Ce code est converti en utilisant une bibliothèque contenant toutes les macro-définitions spécifiques à la machine parallèle cible. Le code exécutable est ainsi obtenu.

La figure 2.9 résume l'organisation des différents modules de SKiPPER-I.

2.3.2 L'étape d'expansion des squelettes

Dans cette étape, la formulation fonctionnelle permettant d'établir l'organisation de l'application est traduite en une représentation intermédiaire exploitable par les modules suivants de SKiPPER. Cette représentation met en évidence le parallélisme de l'application. Elle l'explicite sous la forme d'un Graphe Flot de Données Conditionné GFDC [Sor94] [GLS98]. Dans un tel graphe, les nœuds correspondent aux fonctions de calcul de l'application, tandis que les arcs traduisent les dépendances de données entre ces fonctions. La description d'une application fonctionnelle telle que celle donnée ci-dessous (calcul d'un histogramme) en Caml est traduite sous la forme du GFDC représenté à la figure 1.4 page 35.

```
let image       = acq    512;;
let histogramme = scm    8     rowblock histo fushist image;;
let resultat    = affich histo;;
```

8. En fait (cf. section 2.3.6), le mod`ele d'implantation est mixte : statique *et* dynamique.

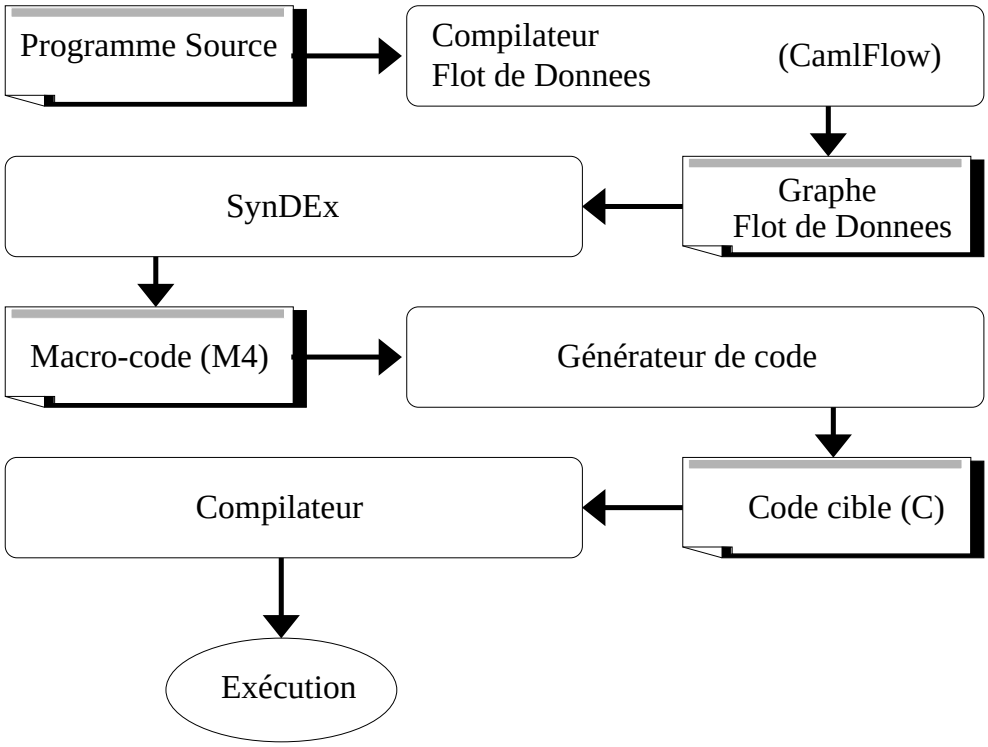


FIG. 2.9 – *L'environnement SKiPPER-I.*

Pour réaliser ce travail, SKiPPER-I utilise le logiciel CamlFlow développé par J. Sérot [Sér01]. Ce logiciel effectue l'analyse syntaxique, l'inférence et le contrôle de types sur le code Caml et produit en sortie une représentation des inter-dépendances relevées dans le programme de l'utilisateur sous la forme d'un GFDC. Pour ce faire, il utilise une technique appelée «interprétation abstraite» [AH87].

Dès lors, SKiPPER-I dispose d'une représentation de l'application à la fois indépendante de l'architecture de la machine cible et des langages de programmation utilisés pour produire le code exécutable.

2.3.3 L'étape de placement et d'ordonnancement

La représentation sous forme d'un GFDC explicite le schéma de calcul/communication des squelettes. Mais il est nécessaire de le mettre en correspondance avec le parallélisme disponible sur la machine utilisée. Cela suppose le placement des fonctions du graphe (sous forme de processus) et des communications respectivement sur les unités de calcul (processeurs) et sur les liens de communications formants le réseau de la machine. L'exécution des processus doit alors être ordonnancée au mieux pour limiter les phases d'attente de données. Cela doit aussi se passer sans risque d'inter-blocage.

Le nombre de combinaisons potentielles à évaluer étant trop grand, l'outil SynDEx qui est en charge de ce travail, fonctionne sur la base d'heuristiques. Un certain nombre de transformations du GFDC sont alors appliquées pour faire coïncider le graphe de l'application (parallélisme potentiel) avec le graphe de l'architecture (parallélisme effectif).

Une fois la correspondance établie, SynDEx produit un macro-code (M4) qui séquence les différentes opérations de calcul et de communication. A ce niveau, l'utilisation de macro-

instructions rend le programme indépendant de l'architecture cible dans la mesure où les macro-instructions de gestion (mémoire, prise de décision (tests),...) et de communication ne sont pas encore traduites. Leur implantation dépendra par la suite de l'architecture. Cette représentation est donc indépendante de l'architecture dans les sens où elle s'affranchit du type de processeur et des caractéristiques physiques du type de réseau. Cependant elle n'est pas indépendante de la topologie de la cible. En effet, elle tient compte de l'agencement du réseau pour calculer les routages adéquats et qui seront physiquement réalisés sur l'architecture cible dans la phase suivante. Cette façon de procéder permet de produire un code fidèle à ce que sera le code exécutable final sans qu'il soit lié à une machine particulière. La seule chose dont dépend le macro-code est donc le nombre de processeurs disponibles sur la machine cible et la topologie du réseau utilisé.

SynDEx a été choisi pour prendre en charge cette partie du travail car il semblait fastidieux de devoir créer de toute pièce ce genre d'outil. Ses fonctionnalités ayant déjà été éprouvées [Gin95], elles ont été mises à profit dans le cadre de SKiPPER-I.

2.3.4 L'étape de génération du code cible

Le macro-code obtenu à l'étape précédente est entièrement traduit en langage C grâce au macro-processeur standard M4. Les macro-instructions spécifiques à la machine cible viennent alors prendre la place des différentes macro-instructions.

Le code C n'a plus qu'à être compilé.

Il faut noter que la plus grande partie du code généré est statique : les étapes du séquencement des calculs et des communications sont décidées à l'étape précédente. Au moment de la génération du code cible, le séquencement des opérations est complétement déterminé et ne peut plus être changé que par d'une nouvelle étape de placement/ordonnancement suivie d'une phase de recompilation.

2.3.5 Apports de SKiPPER-I

SKiPPER-I propose un environnement répondant aux exigences du prototypage rapide et offrant une bonne efficacité pour l'implantation d'applications de vision de complexité réaliste et devant opérer sur un flot vidéo. Cela a notamment été montré dans [GSD98] (segmentation par Etiquetage en Composante Connexe), [SGD99] [GSDC99] (suivi de véhicules par amers visuels) et [SGCD01] (suivi de route).

De la phase de spécification des applications jusqu'à la génération du code cible, l'environnement gère automatiquement les différentes phases de production de l'exécutable, libérant ainsi le programmeur de toutes les tâches complexe de mise en œuvre du parallélisme. Cela tend bien évidemment à réduire les sources d'erreurs et participe de ce fait à l'accélération du cycle de développement allant de la conception à la validation d'une application de vision. La méthodologie du prototypage rapide exploitée par SKiPPER-I autorise ainsi l'utilisateur à tester rapidement plusieurs solutions algorithmiques.

En outre, l'utilisation de l'outil SynDEx pour le placement et l'ordonnancement permet la production d'un exécutif distribué, optimisé et sûr, par des heuristiques prenant en compte les possibilités éventuelles de recouvrement des communications par les calculs.

L'emploi de SynDEx a aussi offert une première solution au problème de portage des applications d'une plate-forme matérielle à une autre : l'effort de portage se limite au développement des macro-définitions de l'exécutif.

2.3.6 Limitations de SKiPPER-I

2.3.6.1 La question de l'imbrication

La principale limitation de SKiPPER-I tient dans l'impossibilité de combiner de manière systématique plusieurs squelettes dans une même application autrement qu'en les séquençant. Elle ne permet pas en particulier l'imbrication. En effet, hormis les squelettes SCM [9] et ITER-MEM, aucun des autres (les squelettes dynamiques) n'autorise l'utilisation d'un autre squelette comme «fonction» de calcul. L'utilisateur ne peut donc pas construire librement un schéma de parallélisation plus complexe que ceux disponibles immédiatement avec les trois squelettes qui lui sont proposés autrement qu'en les enchaînant séquentiellement.

L'imbrication de squelettes est la composition la plus difficile à traiter car elle rend les squelettes inter-dépendants et crée une hiérarchie entre eux. Bien que le besoin de disposer de possibilités d'imbrication n'ait jamais été démontré définitivement [Col99] [10], il apparaît que l'augmentation régulière de la complexité des applications pourrait contribuer à le faire émerger. C'est la cas par exemple de l'application de suivi de visages que nous présentons au chapitre 5.

Qui plus est, le problème de l'imbrication est toujours apparu comme un défi à relever du fait de sa complexité d'implantation pour la communauté de chercheurs s'intéressant aux squelettes algorithmiques. En fait, même dans le cas où l'imbrication ne serait pas réellement nécessaire, il peut s'avérer qu'elle soit malgré tout utile dans la spécification du parallélisme par l'utilisateur. En effet, considérons le cas où, en programmation impérative séquentielle, on est en présence d'une fonction récursive. On peut choisir de l'écrire telle qu'elle, ou de la «dé-récursifier». Si la seconde solution est généralement plus efficace lors de l'exécution, il n'en reste pas moins que son obtention peut être fastidieuse et source d'incompréhension. De même, il peut s'avérer que le schéma de parallélisation d'un algorithme se trouve plus efficacement décrit par le choix d'une imbrication de squelettes plutôt que par l'obtention d'une combinaison équivalente mais non imbriquée. Une autre justification à l'imbrication de squelettes provient de la manière dont est construit l'environnement d'aide à la programmation parallèle. Bien souvent il ne donne pas la possibilité à l'utilisateur de créer de nouveaux squelettes. C'est le cas notamment de SKiPPER. L'utilisateur doit se contenter de ceux proposés dans la bibliothèque de l'environnement, qui est figée. Dans ce cas, il peut être nécessaire de combiner ces squelettes afin d'obtenir un schéma de parallélisation plus complexe que ceux proposés individuellement par chaque squelette mais qui rend mieux compte du parallélisme potentiel de l'application. C'est d'ailleurs aussi le cas avec des environnements extensibles en termes de squelettes (comme Proteus [NP92], FrameWorks [SSS98] ou PASM [GSP98] [GSP99] [GSP00] [GSP01]). Dans ce dernier cas, il peut être plus facile, voire plus efficace, de combiner plusieurs squelettes entre eux pour en former un «nouveau» plutôt que de programmer un nouveau schéma *ex nihilo*.

9. Le squelette SCM autorise l'imbrication d'un autre squelette du mˆeme type car sa d´efi nition est suffi sante pour produire un Graphe Flot de Donn´ees Conditionn´e valide pour SynDEx.

10. Tout particuli`erement dans le cadre d'applications de vision.

2.3.6.2 Origine du problème

La principale raison aux limitations de SKiPPER-I en termes de composition de squelettes tient à l'usage de l'outil SynDEx comme dorsal («*backend*»). Ce choix parfaitement justifié pour les premiers squelettes introduits dans SKiPPER (le SCM en particulier qui possède une interprétation immédiate sous la forme d'un GFDC [GSD98]) conduit à un problème de définition de squelettes qui s'expriment plus naturellement sous la forme d'un *graphe de processus*. C'est le cas en particulier de ceux fondés sur la mise en œuvre d'une ferme de processeurs (DF et TF). Le modèle d'exécution de ces squelettes est par essence dynamique : l'ordonnancement de la plupart des communications qu'ils mettent en jeu ne peut être décidé qu'à l'exécution. Or SynDEx ne peut mettre en œuvre que des modèles d'exécution statiques, les seuls qui peuvent être formulés sous la forme d'un GFDC.

Pour contourner cette difficulté, D. Ginhac [Gin99] confine les communications dynamiques des squelettes DF et TF entre deux barrières de synchronisation dans le GFDC. Elles sont alors «masquées» du point de vue de SynDEx. Cet artifice donne alors toute liberté de les implémenter pour les intégrer à l'exécutif produit par SynDEx.

Cette solution soulève toutefois plusieurs problèmes [CSD00].

Premièrement, SynDEx ne peut plus gérer la partie de l'application qui a été plaçée entre ces barrières de synchronisation. Donc toutes les communications et l'ordonnancement des processus des squelettes DF et TF échappent à son contrôle. Il n'est alors plus en mesure d'optimiser cette partie de l'application.

Deuxièmement, des communications «étrangères» à celles ordonnancées statiquement par SynDEx sont introduites volontairement. Il faut donc veiller à ce qu'elles n'interfèrent pas avec les mécanismes de communication natifs de SynDEx. C'est d'ailleurs la raison pour laquelles des barrières de synchronisation sont mises en place. Elles ont pour objet de confiner toutes les opérations extérieures à celles gérées par SynDEx. Tant que l'imbrication de squelettes n'est pas autorisée, cette manière de procéder se révèle efficace et ne pose pas de problèmes d'interactions entre les deux gestions des communications (l'une statique par SynDEx, l'autre dynamique par les processus auxiliaires dédiés des squelettes DF et TF [11]). Mais le problème devient plus difficile à traiter lorsque l'imbrication de deux squelettes est permise. En effet, imbriquer un squelette SCM dans un squelette DF par exemple revient alors à interdir à SynDEx de gérer le squelette SCM (puisque masqué). Aucun des modules de SKiPPER-I n'est prévu pour pallier ce manque.

Troisièmement, la description du schéma de parallélisation des squelettes DF et TF, après ce que nous venons d'en dire, ne peut être faite que de manière *ad hoc*, c'est-à-dire qu'elle est spécifique à la machine cible : elle ne peut être fournie sous forme d'un GFDC indépendant de la cible. Pour ces deux squelettes, il n'existe donc pas de représentation intermédiaire indépendante de la machine cible. Qui plus est, ce dernier point donne une spécification des squelettes qui n'est pas homogène, ce qui entraîne une difficulté de principe dans la mise en œuvre d'une composition de ces squelettes, encore plus pour une imbrication.

11. Ces processus prennent en charge les communications dynamiques li´ees au fonctionnement interne des squelettes dynamique qui ne peut ˆetre directement d´ecrit avec SynDEx, `a savoir : envoi des donn´ees `a traiter du maˆıtre aux esclaves et des r´esultats interm´ediaires des esclaves vers le maˆıtre. Ils sont «encapsul´es» dans les fonctions associ´ees aux nœuds du GFDC.

2.3.6.3 Une première approche du problème sous SKiPPER-I

Pour solutionner le problème de la représentation statique de squelettes dynamiques sous SynDEx à l'aide d'un GFDC, nous avons proposé en 1999 une voie originale destinée à conserver le fonctionnement de la première version de SKiPPER tout en la préparant à intégrer la notion de composition de squelettes [CSD00]. La représentation d'un squelette dynamique de type DF ou TF est alors la même que celle utilisée pour un SCM, à la différence près qu'on fait apparaître une mémoire d'état liant la distribution des données aux esclaves et leur collecte (voir figure 2.10).

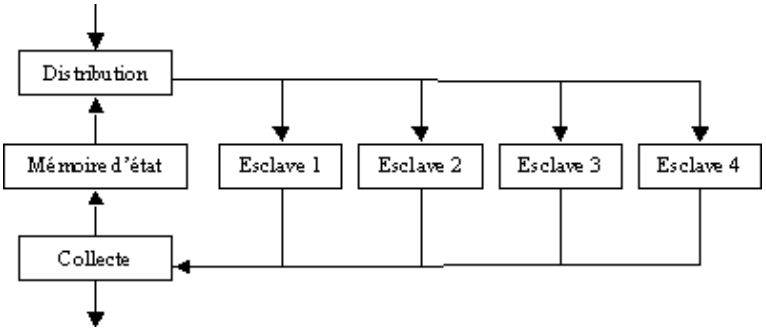


FIG. 2.10 – *Représentation du squelette Data Farming statique.*

Dans ce modèle d'exécution, les fonctions utilisateurs servant d'esclaves ne sont pas exécutées entièrement, mais en partie seulement. Plus précisément elles sont exécutées durant un quota de temps donné (mécanisme similaire à celui utilisé dans les systèmes d'exploitation pour le temps partagé). On exécute donc en boucle le GFDC du squelette jusqu'à ce que toutes les données initiales aient été traitées par les esclaves. Grâce à l'exécution par quota de temps des esclaves, le DF et le TF peuvent être décrits sous SynDEx sans qu'il y ait attente de fin d'exécution d'une fonction de calcul assignée à un esclave pour pouvoir en alimenter un autre. On peut donc considérer les squelettes dynamiques ainsi décrits comme des squelettes SCM à grain fin itérés. L'annexe B et l'article [CSD00] décrivent plus en détail le modèle d'exécution que nous venons brièvement d'exposer.

Malgré le fait que cette approche permettait une représentation statique de squelettes dynamiques – et nous rapprochait ainsi d'une représentation intermédiaire beaucoup plus homogène (à défaut d'être unique) en considérant la représentation des squelettes dynamiques comme une généralisation de tous les autres – elle a dû être abandonnée. L'une des principales raisons est qu'elle nécessitait l'introduction de la notion de boucle d'itération localisée dans le temps sous SynDEx. Cette boucle d'itération temporelle, ou «dépliage temporel», est la possibilité de répéter un nombre de fois déterminé un motif (une fonction, une communication,...) à l'intérieur même d'un GFDC. Cette possibilité étant indisponible dans la version de SynDEx (v4) que nous utilisions, et n'étant pas vouée à être intégrée à court terme [12] dans de futures évolutions du logiciel, nous nous sommes tournés vers une autre approche (cf. chapitre 4).

12. Cette fonctionnalit´e n'est apparue que mi-2002 dans la version v6.

2.3.7 Justification de l'évolution de SKiPPER

Les limitations précédentes nous ont donc conduit à proposer un autre modèle d'exécution pour les squelettes de SKiPPER. L'environnement de programmation bâti sur ce nouveau modèle (SKiPPER-II) (et que nous présentons dans ce mémoire au chapitre 4) autorise la composition des squelettes de SKiPPER, en plus du séquencement, et tout particulièrement les cas d'imbrication. Le séquencement de squelettes étant déjà disponible sous SKiPPER-I, nous avons mené une nouvelle étude sur le cas plus difficile de l'imbrication. L'imbrication peut être formulée sous la forme :

```
let nest outer inner x = outer inner x
(* avec outer et inner, deux squelettes : *)
(*   outer: (a -> b) -> a -> c           *)
(*   inner: a -> b                       *)
```

Voulant proposer un environnement autorisant la combinaison de squelettes et étant confrontés au fait que certains squelettes (DF et TF) ne disposaient pas d'une représentation intermédiaire indépendante de l'architecture cible, nous en sommes arrivés à proposer une version de SKiPPER qui tienne compte de ce point. Pour ce faire, nous avons développé une représentation unique pour tous les squelettes qui soit à même d'intégrer toutes les fonctionnalités de tous les squelettes. Cette représentation est fondée sur la description d'un «méta-squelette», appelé TF/II, et qui synthétise le comportement potentiel de n'importe quel autre squelette de notre bibliothèque. Sa représentation intermédiaire est totalement indépendante de toute architecture cible.

Avant d'aborder la description détaillée de cette nouvelle version de SKiPPER au chapitre 4, nous présentons au chapitre suivant une revue des environnements de programmation les plus significatifs autorisant l'imbrication de squelettes algorithmiques.

Chapitre 3

Le problème de l'imbrication de squelettes et outils la supportant

L'état-de-l'art qui suit se concentre sur les environnements de programmation parallèle offrant une méthodologie fondée sur l'emploi des squelettes algorithmiques. Qui plus est il se restreint à ceux qui autorisent la composition de ces squelettes, plus particulièrement au sens de l'imbrication. Le lecteur désireux d'obtenir une vue plus large d'environnements développants l'utilisation des squelettes algorithmiques pourra se reporter aux thèses de D. Ginhac [Gin99] et de M.M. Hamdan [Ham00]. Parallèlement à cela, nous abordons aussi la question des formes d'imbrication licites sous SKiPPER.

3.1 Introduction

Ce chapitre présente à travers les environnements les plus significatifs que sont SCL, P^{3}L, HOPP, EKTRAN et SML, les différentes approches retenues pour traiter le problème de l'imbrication de squelettes.

L'imbrication est à la fois un problème mathématique si on l'aborde du côté de la modification du graphe des squelettes, et un problème informatique si on l'aborde du côté de l'implantation directe sur machine.

Mais elle est avant tout un élément de méthodologie. En effet, la mise à disposition de cette possibilité pour l'utilisateur lui permet de découper son application en plusieurs niveaux de parallélisation qu'il peut considérer de manière indépendante. Ainsi il peut se concentrer sur un seul niveau à la fois, et non pas sur la parallélisation globale de son application. L'imbrication, et la composition de squelettes d'une façon plus générale, élève donc le niveau d'abstraction potentiel proposé par un environnement de programmation parallèle par squelettes.

3.2 Travaux de Darlington *et coll.*

Ce groupe de travail de l'Imperial College de Londres a très fortement contribué à la notoriété des squelettes algorithmiques à travers la communauté depuis le début des années 90.

Dans ces travaux, l'expression des squelettes algorithmiques se fait à l'aide de fonctions d'ordre supérieur [1] dans un langage de type fonctionnel [DGT93], comme il est proposé dans de nombreux environnements de programmation, et notamment dans SKiPPER. Concernant l'implantation effective des squelettes, les auteurs font remarquer que certains de leurs squelettes sont plus adéquats pour certaines architectures cibles que d'autres, même s'ils peuvent tous être implantés sur n'importe quelle cible. De ce fait, ils proposent des moyens de transformation de certains squelettes en d'autres afin d'obtenir une implantation plus efficace en fonction de la machine cible. C'est une approche originale par rapport aux autres environnements, et notamment SKiPPER-II où nous suggérons que la meilleure implantation de nos squelettes pour une machine cible donnée est garantie par l'utilisation d'une bibliothèque de communication standard (MPI), mais où l'adéquation du schéma du squelette à l'architecture cible (par rapport à un autre) n'est pas prise en compte automatiquement ; cette dernière ne l'est que dans la mesure où SKiPPER est un outil d'*aide au prototypage rapide*, et de ce fait permet d'évaluer rapidement l'intérêt de plusieurs squelettes face à un problème donné, et où le programmeur peut demander conseil à des experts du domaine pour l'assister.

3.2.1 La bibliothèque de squelettes

La bibliothèque de squelettes définie à l'Imperial College en comporte quatre, certains conçus pour le parallélisme de données, d'autres pour le parallélisme de tâches. Le langage fonctionnel utilisé pour leur déclaration est Haskell [HPF99b] [HPF99a].

1. cf. annexe A.

Les squelettes retenus sont :

- PIPE,
- FARM,
- DC,
- RaMP.

Le squelette PIPE exploite directement la technique du *pipeline* pour réaliser un parallélisme de tâches. A chaque processeur est assigné une ou plusieurs tâches. La définition (en Haskell), ou sémantique déclarative, de ce squelette est la suivante :

$$PIPE :: [[\alpha] \to [\alpha]] \to [\alpha] \to [\alpha]$$

$$PIPE = foldr\ (\circ)$$

Le squelette FARM décrit le comportement classique d'une ferme de processeurs (parallélisme de données). Sa définition est la suivante :

$$FARM :: (\alpha \to \beta \to \gamma) \to \alpha \to ([\beta] \to [\gamma])$$

$$FARM\ f\ env = map\ (f\ env)$$

Le squelette DC correspond au schéma *Divide and Conquer*. Dans celui-ci, les tâches initiales sont récursivement découpées en sous-tâches. Le découpage se poursuit jusqu'à obtenir des tâches «élémentaires». Ces dernières sont exécutées et leurs résultats sont fusionnés pour produire le résultat final du squelette. Sa définition est alors la suivante :

$$DC :: (\alpha \to Bool) \to (\alpha \to \beta) \to (\alpha \to [\alpha]) \to ([\beta] \to \beta) \to \alpha \to \beta$$

$$DC\ t\ s\ d\ c\ x$$

$$\quad | \ t\ x = s\ x$$

$$\quad | \ not\ (t\ x) = (c \circ map\ (DC\ t\ s\ d\ c) \circ d)\ x$$

Le nom du squelette RaMP signifie quant à lui : *Reduce and Map over Pairs*. Il est utilisé dans les situations où il faut gérer deux listes d'éléments, et où chacun des éléments d'une liste peut inter-agir avec n'importe lequel de l'autre (exemple applicatif : le problème des N corps en physique). RaMP a pour définition :

$$RaMP :: (\alpha \to \beta \to \gamma) \to (\gamma \to \gamma \to \gamma) \to [\alpha] \to [\beta] \to [\gamma]$$

$$RaMP\ f\ g\ as\ bs\ = map\ h\ as\ where\ h\ x = foldr1\ g\ (map\ (f\ x)\ bs)$$

3.2.2 Squelettes spécifiques à un domaine applicatif

Les squelettes de la bibliothèque du projet que nous avons mentionnés sont des squelettes à usage général. Il ne sont pas dédiés à un domaine applicatif précis. Cependant le groupe de Darlington a introduit dans ses travaux [DT93] la notion de squelettes dédiés à un domaine applicatif donné. Le domaine d'application choisi alors est celui de la synthèse d'images avec le modèle géométrique CSG (Constructive Solid Geometry). Deux squelettes y sont définis : *transformCSG* et *reduceCSG*. Ils correspondent respectivement à un `map` et à un `fold` appliqués sur l'arbre de représentation CSG.

Le projet SKiPPER a suivi cette approche en proposant une bibliothèque de squelettes spécialement conçus pour le traitement d'images [2].

3.2.3 SCL

L'aboutissement des travaux du groupe de l'Imperial college est SCL, pour *Structured Coordination Language* [DGTY95b] [DGTY95a]. Avec ce langage, un système de programmation à deux niveaux est introduit. Le haut-niveau est un modèle de coordination et le bas-niveau un modèle pour la partie calculatoire, pouvant intégrer n'importe quel langage séquentiel classique. Le modèle de coordination se compose de squelettes algorihmiques décrit en langage fonctionnel. Les squelettes sont ici réservés à la spécification des schémas de parallélisation utilisés pour décrire un algorithme. Ils permettent de décharger le programmeur des problèmes de placement et d'ordonnancement des tâches et des communications. En cela, cette approche est similaire à celle de SKiPPER.

L'intérêt donné par les auteurs pour une telle approche est principalement la possibilité élevée de réutilisation de code ; en composant des modules de code déjà existants avec ce langage de coordination, leur réutilisation s'en trouve facilité. La portabilité et le maintien d'un certain niveau de performance sur des plate-formes différentes sont supposés être obtenus par un ensemble d'implantation des squelettes spécifique à tout un jeu d'architectures cibles.

Les squelettes faisant partie de SCL sont des squelettes non spécifiques à un domaine. Ils sont rangés dans trois catégories :

- squelettes de configuration,
- squelettes élémentaires
- squelettes de traitement.

La catégorie *configuration* fait intervenir trois squelettes. Ce sont des squelettes de partitionnement et de placement de données indépendamment de l'architecture cible. Ils servent essentiellement à préparer les données en vue de leur traitement par les squelettes des deux autres catégories. En ce sens, ils ne correspondent pas complètement à la même vue des squelettes que ceux des deux autres catégories.

2. Mˆeme si l'exp´erience a montr´e, *a posteriori*, que les squelettes de SKiPPER avaient en fait une port´ee plus g´en´erale.

Les squelettes de configuration sont :

- *partition*, qui cré un tableau distribué des données à partir d'un centralisé,
- *align*, qui regroupe de données placées dans des tableaux distribués pour qu'ils correspondent à l'architecture cible,
- *gather*, qui recombine les données de tableaux distribués en un tableau séquentiel.

Les squelettes *élémentaires* réalisent un certain nombre d'opérations mettant en œuvre un parallélisme de données sur les tableaux distribués fournis par les squelettes de la catégorie précédente. Les principaux squelettes de cette catégorie sont de type `map`, `fold` et `scan`, ainsi que le squelette *imap* qui prend en plus comme argument un tableau d'index.

Enfin, les squelettes de la catégorie *traitement* encapsulent deux types de parallélismes communément utilisés : une ferme de processeurs avec le squelette *farm*, et le modèle SPMD avec le squelette du même nom. De plus, deux autres squelettes viennent s'ajouter à cette liste pour prendre en compte des structures répétitives. Ce sont les squelettes *iterUntil* et *iterFor* qui, respectivement, utilisent une condition et un compteur pour contrôler la fin de leur exécution. Ces squelettes se rapprochent de ceux utilisés dans SKiPPER, les squelettes de configuration ne se retrouvant pas dans notre projet puisque les fonctions de partition de données sont fournies par l'utilisateur (fonctions *split* du SCM et *divide* du TF, par exemple).

L'imbrication de squelettes est proposée dans SCL par introduction de squelettes dans le squelette SPMD. Elle repose sur une transformation des squelettes avant l'exécution, et donc par une mise «à plat» de l'imbrication demandée par le programmeur. L'imbrication est ici gérée à la compilation, plutôt qu'à l'exécution comme avec SKiPPER-II. On notera que l'imbrication, nécessitant ici une transformation des squelettes avant exécution, n'est prise en charge que pour un seul squelette (SPMD), contrairement à SKiPPER-II qui supporte l'imbrication pour l'intégralité de son jeu de squelettes. La contrepartie est que l'implantation résultante avec une approche de l'imbrication à la compilation peut se révéler plus efficace en termes de performances qu'une approche à l'exécution, surtout si le nombre de combinaisons entre squelettes est faible.

3.3 P^{3}L

Le projet P^{3}L ayant été décrit à la section 1.5.3.1 page 48, nous nous intéressons ici à ses capacités d'imbrication de squelettes.

L'imbrication est autorisée, au sens de P^{3}L. Elle est à la charge du programmeur qui la spécifie explicitement à l'aide des constructeurs parallèles du langage qui lui sont proposés. Cependant, les documents relatifs au projet font peu de cas de la manière dont le compilateur résout le problème de l'imbrication. La complexité de la tâche est cependant limitée par l'obligation qu'a le programmeur de spécifier explicitement les relations de dépendance entre les différents squelettes.

Il faut noter en outre que, parmi les squelettes algorithmiques proposés, seuls *sequential*, *pipe*, *farm* et *loop* sont autorisés pour l'imbrication. La composition des squelette est de type hiérarchique [Pel97].

3.4 HOPP

HOPP (Higher-Order Parallel Programming model) est un langage fonctionnel à parallélisme implicite [Ran95] [Ran96]. A chaque squelette est associée une fonction (d'ordre supérieur) et réciproquement [3]. Comme avec les autres systèmes utilisant ce type de formalisme, la fonction d'ordre supérieur permet à l'utilisateur de suggérer le schéma de parallélisation qu'il veut exploiter sans entrer dans les détails de sa mise en œuvre. Ces détails sont pris en charge par l'implémentation de chaque squelette qui est dépendante de la machine cible. Les machines considérées par *HOPP* est de type MIMD-DM. Avec *HOPP* tous les squelettes sont associés à un modèle des coûts de réalisation. Ainsi on peut connaître *a priori* l'efficacité du schéma qu'on souhaite utiliser pour une machine cible donnée. Les fonctions de calcul proposées par l'utilisateur sont forcément séquentielles. Les seules fonctions qui donneront lieu à la production d'un code parallèle sont celles qui sont associées à des squelettes. Les fonctions de calcul de l'utilisateur viennent en paramètre des fonctions d'ordre supérieur, mais leur contenu n'est pas parallélisé.

Le modèle de programmation *HOPP* consiste donc en trois modèles distincts :

– programme,

– machine,

– coût.

Le modèle *programme* est en fait l'ensemble du code source de l'utilisateur composé à la fois de ses fonctions de calcul (séquentielles) et des fonctions d'ordre supérieur associées aux squelettes (`map`, `fold`, `scan` et `filter`) et qui caractérisent l'architecture du programme. Les squelettes peuvent être librement imbriqués, mais on notera que *HOPP* ne parallélise que les trois premiers niveaux d'imbrication seulement.

Le modèle *machine* est un ensemble de machines cibles potentielles dont l'une sera instanciée lors de la génération du code cible (en langage C).

Enfin le modèle *coût* correspond aux coûts de mise en œuvre de chaque squelette sur une architecture donnée, comme nous l'avons mentionné précédemment. Le système complet donne à la fois une méthodologie et des modèles de coûts, mais aussi un programme d'analyse de ces coûts de fonctionnement pour les estimer à la compilation.

La figure 3.1 présente le synoptique du modèle *HOPP*. L'analyseur construit une représentation du programme sous la forme d'un arbre, exploitable par les étages suivants. Chaque branche de l'arbre représente une phase du programme. Une analyse des coûts pour chaque branche est ensuite réalisée. Le coût d'une branche dépend non seulement de la nature et du nombre de squelettes qu'elle contient, mais aussi de l'architecture cible retenue, pour un nombre de processeurs donné. Le générateur de code peut alors produire l'exécutable pour l'architecture retenue. Il est à noter que la phase de génération du code cible n'est pas réalisée de manière complètement automatique.

3. Dans les approches implicites on rencontre le plus souvent la relation suivante : `a une fonction d'ordre sup´erieur sont potentiellement associ´es plusieurs squelettes.

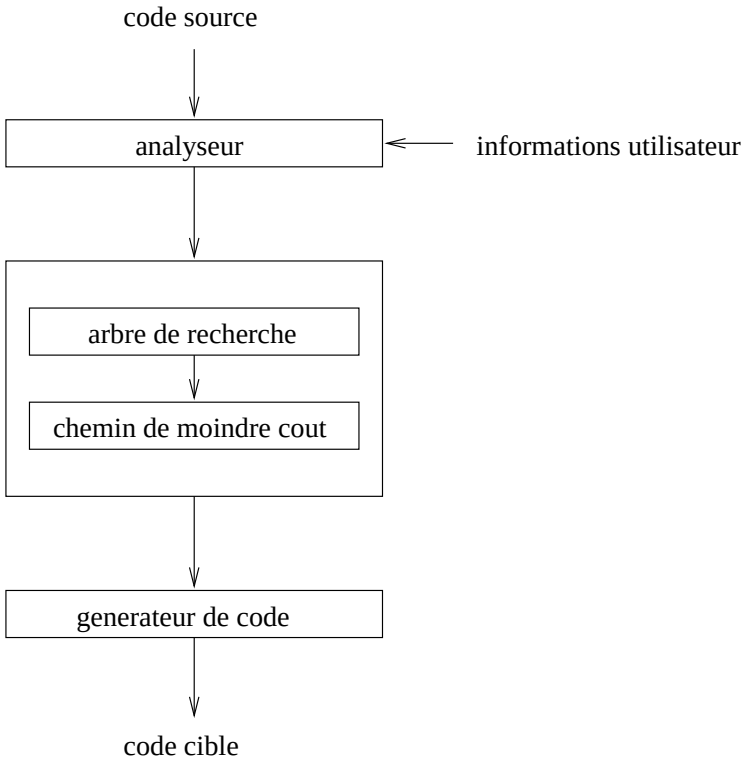


FIG. 3.1 – *Synoptique de HOPP (d'après [Ran95]).*

3.5 EKTRAN

EKTRAN est un langage fonctionnel simple, développé à l'université Heriot-Watt d'Edimbourg en Ecosse [Ham00], et intégrant de manière native la notion de squelette algorithmique comme implantation directe de certaines fonctions du langage. Il est ainsi un support de programmation parallèle utilisant les squelettes algorithmiques.

Un programme parallèle peut ainsi être construit en écrivant un programme fonctionnel normal, mais en utilisant des fonctions d'ordre supérieur admettant une formulation parallèle (squelette) aux endroits du code où le programmeur souhaite exploiter le parallélisme. L'implantation faite des squelettes est une implantation complètement dynamique, dans le sens où placement et ordonnancement des calculs et des communications sont réalisés intégralement à l'exécution. Le support de l'imbrication est fourni par la méthode même d'implantation des squelettes. Les squelettes sont mis en œuvre par l'intermédiaire des fonctions évoluées de MPI et notamment la notion de *groupes de processus*. A chaque squelette est attribué un groupe MPI. Ainsi toutes les communications internes à un squelette se font au sein de son groupe. Elles ne peuvent ainsi venir perturber les communications d'un autre. Les opérations de chaque squelette peuvent de ce fait être isolées de celles des autres squelettes. Ces derniers peuvent ainsi cohabiter sur le réseau de processeurs sans qu'il se produise de collisions dans leurs traitements, les seules interactions pouvant avoir lieu étant celles marquant leurs dépendances éventuelles. L'imbrication potentielle de squelettes découle directement de ce mécanisme.

3.5.1 La bibliothèque de squelettes

Les squelettes définis dans EKTRAN sont au nombre de trois :

- `map`,
- `fold`,
- `compose`.

Ils correspondent directement aux fonctions d'ordre supérieur du même nom du langage. Ce sont des squelettes à usage général et non dédiés à un domaine applicatif, contrairement à ceux proposés dans le projet SKiPPER.

Le squelette `map` (parallélisme de données) applique une fonction à chaque élément de la liste qui lui est fournie en argument. Sa définition est la suivante : $map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, ..., f\ x_n]$; soit en Caml :

```
> let map f [] = [] | map f (h::t) = (f h) :: map f t

# map : ('a -> 'b) -> 'a list -> 'b list
```

Son implantation en tant que squelette fonctionne schématiquement de la sorte. Trois processus sont mis en œuvre pour faire fonctionner ce squelette :

- un processus de supervision du squelette,
- un processus esclave,
- un processus esclave spécial pour les cas d'imbrication.

Le processus de supervision se charge de créer les sous-groupes nécessaires à une éventuelle prise en charge d'une imbrication en son sein, et de réaliser les allocations de processeurs. Ensuite, il exécute l'algorithme esclave soit sous la forme du processus esclave s'il n'y a pas d'imbrication, soit sous celle du processus esclave spécial en vue de l'imbrication. Ce dernier a pour rôle de gérer la relation du squelette avec le squelette imbriqué après l'avoir démarré. Dans son approche dynamique, SKiPPER-II se distingue par l'absence de processus spéciaux de gestion des imbrications. Contrairement à EKTRAN, il ne fait aucune différence entre la mise en œuvre d'un squelette imbriqué et celle d'un squelette non imbriqué : aucun processus supplémentaire, ou spécifique, n'a besoin d'être activé pour gérer, ou communiquer avec, un squelette imbriqué. De plus, la technique d'imbrication utilisée par EKTRAN s'appuie sur les mécanismes MPI pour cloisonner les squelettes qui sont imbriqués. De ce fait, elle concède une partie de la gestion de l'imbrication à MPI. Par rapport à SKiPPER-II, ce choix présente le désavantage de nécessiter plus de fonctionnalités de la norme MPI (portage moins aisé sur des architectures dédiées). Qui plus est, il occasionne aussi un léger surcoût dans la gestion des communciations par la bibliothèque (tri des communications).

Le squelette `fold` applique une fonction sur la liste qui lui est donnée en argument suivant la définition : $fold\ f\ b\ [x_1, x_2, \ldots, x_n] = f\ x_1\ (f\ x_2\ (\ldots f\ x_n\ b)))$; soit en Caml :

```
> let fold f b [] = b | fold f b (h::t) = f h (fold f b t)

# fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

Ce squelette est implanté sous la forme d'un schéma de type *divide and conquer* binaire. Ainsi, le processus maître découpe la liste initiale en deux sous-listes, dont une est transmise à un processus esclave et l'autre est conservée pour traitement immédiat [4]. Le processus esclave répète la procédure.

Le squelette `compose` est conçu comme une *pipeline* (parallélisme de tâches).

3.5.2 La gestion de l'imbrication

Avec EKTRAN, l'imbrication de squelettes est supportée jusqu'à n'importe quel niveau de profondeur. Le mécanisme est directement intégré dans celui de gestion de chaque squelette. Il est mis en œuvre en exploitant les fonctions évoluées de MPI de gestion des *groupes de communication*. L'idée qui guide le mécanisme est de procéder à l'allocation des processus nécessaires aux squelettes à des sous-groupes MPI au fur et à mesure que des squelettes imbriqués sont identifiés dans les fonctions d'ordre supérieur, en commençant par celle la plus «englobante» (de plus au niveau). Plus précisément, lorsqu'un programme commence son exécution, seul un unique groupe MPI existe. Il contient déjà la totalité des processus (non encore instanciés) qui pourront servir à l'exécution des squelettes. Lorsqu'il y a nécessité d'imbrication, alors un nouveau sous-groupe est activé pour le squelette imbriqué. Le squelette englobant réalloue les processus de son groupe de manière à ce que le squelette imbriqué puisse s'exécuter. Ce mécanisme est corrélé à l'utilisation des groupes MPI. C'est une différence de fonctionnement importante avec SKiPPER-II. Dans notre cas, les ressources appartiennent à un squelette et ce dernier, en cas d'imbrication en son sein, n'a pas à en libérer pour les donner à un autre. Le squelette imbriqué «se sert» dans les ressources encore libres. On évite ainsi le surcoût induit par l'allocation (en quelque sorte) puis la réallocation successive d'une même ressource à deux squelettes différents. De plus, avec SKiPPER-II, les ressources ne sont pas allouées définitivement à un squelette, mais en fonction de ses besoins au cours du temps. Ainsi, s'il n'a plus besoin de certaines d'entre elles, elles peuvent être libérées au fur et à mesure ; un autre squelette peut alors se les approprier. EKTRAN met de plus en place un groupe de supervision (reliant tous les sous-groupes crées) pour les relier et assurer leur gestion (voir figure 3.2). Ce processus se poursuit de manière récursive formant un arbre de sous-groupes MPI. Là encore, par sa technique d'imbrication, SKiPPER-II s'affranchit de processus de gestion spécifiques : les seuls processus à être exécutés font partie intégrante des squelettes et de leurs schémas. Dans SKiPPER-II, les ressources sont entièrement dédiées aux squelettes, et ces derniers sont auto-suffisants pour leur propre gestion et celle des squelettes qui leur sont imbriqués.

3.6 Travaux de Michaelson et coll.

Les travaux de Michaelson et coll. ayant déjà fait l'objet d'une présentation à la section 1.5.3.2 page 50, nous ne nous intéressons ici qu'aux possibilités d'imbrication.

Ce compilateur autorise l'imbrication de squelettes, qui peut être faite pour une profondeur indéterminée, en ce sens qu'elle n'est pas fixée *a priori* par l'implémentation du compilateur elle-même. L'imbrication est obtenue lorsque une fonction d'ordre supérieur ayant un équivalent squelette est plaçée comme argument d'une autre fonction de ce type. Il est important de noter que l'imbrication de deux squelettes écrite, et donc sollicitée, par l'utilisateur au niveau

4. La fonction f est suppos´ee associative et commutative.

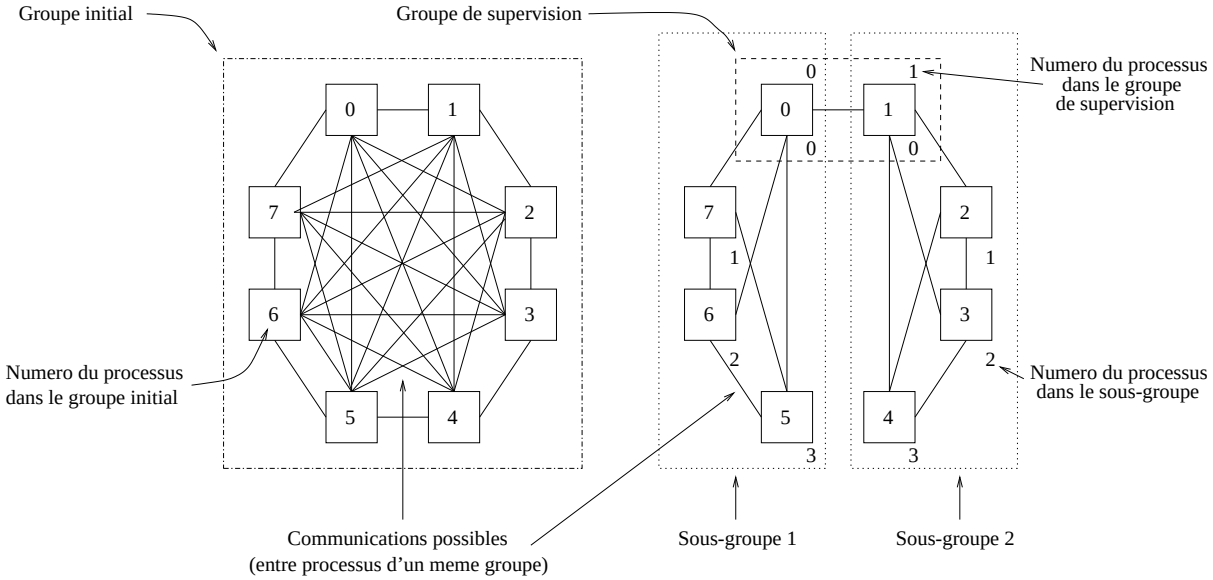


FIG. 3.2 – *Exemple de division d'un groupe MPI en sous-groupes.*

de son code source ne se traduira pas forcément par l'imbrication effective de ces deux squelettes au moment de l'exécution. En effet, le compilateur est capable de choisir entre maintenir l'imbrication ou non (c'est-à-dire d'instancier ou non la fonction d'ordre supérieur «interne» en squelette) en fonction des informations données par le module chargé de faire du «profiling» [5]. L'imbrication ne sera maintenue que si le gain en temps d'exécution justifie la surcharge en communications due à l'imbrication elle-même.

Il faut noter enfin que le compilateur a été construit en grande partie en utilisant des outils logiciels déjà existants (analyse syntaxique, générateurs de code,...). Seules les parties spécifiques au parallélisme ont été développées.

Une des limitations annoncée par les concepteurs [MSBK00] est la surcharge due à la gestion de certaines communications sous MPI. C'est le cas par exemple lors de diffusions de type «broadcast» qui sont implémentées dans l'environnement par des boucles utilisant des fonctions *Send* alors que la fonction native spécialisée de MPI aurait pu, potentiellement, donner de meilleurs résultats. Le problème est que cette dernière reste limitée à un seul communicateur. On notera aussi que les exemples fournis font apparaître des imbrications «artificielles» dont le seul but est d'évaluer le comportement du compilateur (les algorithmes choisis ne nécessitent pas réellement d'imbrication pour leur parallélisation). On note enfin l'absence de baisse significative de performance entre les mêmes algorithmes pris dans leurs versions imbriquée et non imbriquée. Cela n'est d'ailleurs pas une surprise puisque les schémas de parallélisation utilisés sont réguliers, ce qui est le cas dans les exemples de [SBMK98]. En effet, dans ces algorithmes, l'imbrication n'était pas une nécessité en soi, les traitements ne sont pas répartis différemment dans la version imbriquée (l'imbrication sert essentiellement à distribuer et opérer des partitions différentes sur les données initiales que ne l'aurait fait une parallélisation sans imbrication). Il s'en suit que la version imbriquée n'ajoute qu'un surcoût initial, mais aucun en terme de traitement effectif supplémentaire des données du problème. Les auteurs notent aussi que l'imbrication de squelettes représente essentiellement un défi dans la gestion et la coordination des squelettes lorsqu'il faut optimiser la distribution des données entre les processeurs.

5. Le module de «profi ling» est bas´e sur le kit de d´eveloppement *ML Kit evaluator* dans lequel est pass´e un jeu de tests connu pour r´ealiser les mesures d'effi cacit´e [Bra95].

3.7 Etude des formes licites d'imbrications sous SKiPPER

Le problème de l'imbrication de squelettes algorithmiques, même s'il est essentiellement un problème de mise en œuvre, peut aussi être appréhendé d'un point de vue purement sémantique en se posant la question : quelles sont les formes «licites» d'imbrication ?

Cette question des formes licites d'imbrication est à se poser avant toute réalisation éventuelle. En effet, selon la base de squelettes choisie, les possibilités d'imbrication varient : certains squelettes ne pourront pas être imbriqués. Cette impossiblité ne résulte pas simplement d'une imcompatibilité entre les implantations de ces squelettes (bien qu'elle soit aussi à prendre en compte, cf. le chapitre 4). Elle résulte plutôt d'une étude amont permettant d'évaluer les combinaisons potentielles de squelettes qui auront un sens au regard du fonctionnement instrinsèque du squelette englobant lui-même. La difficulté croît avec le nombre de combinaisons possibles et donc avec le nombre de squelettes de la base.

Nous présentons ci-dessous les formes licites d'imbrication des squelettes de la base de SKiPPER, étant bien entendu qu'une forme *licite* ne signifie pas forcément une forme *utile*. Nous entendons par forme licite, une expression valide syntaxiquement. Par utile, nous indiquons une forme soit communément exploitée, soit conduisant à une implantation efficace. Seuls les squelettes SCM, DF et TF sont pris en compte. Le squelette ITERMEM étant un squelette particulier de la base – non appelé à intervenir dans le graphe lui-même des squelettes, exprimant simplement l'itération du graphe – il ne participe jamais à l'imbrication.

Nous procédons comme suit :

- un squelette englobant est fixé,
- pour ce squelette, tous les squelettes de la base sont passés en revue comme fonction de calcul potentielle,
- la validation d'une combinaison possible à un niveau donné d'imbrication valide l'utilisation de sous-imbrications récursives au sein du même squelette.

Ce dernier point est vérifié par le fait qu'une imbrication licite de deux squelettes *ne change pas le comportement du squelette englobant* (sinon elle serait déclarée illicite). Considérons le cas de la figure 3.3 où un squelette 3 est imbriqué dans un squelette 2, qui lui même est imbriqué dans un squelette 1 (le squelette 3 est donc sous-imbriqué dans le squelette 1). Si on suppose l'imbrication du squelette 3 licite dans le squelette 2, et que, comme montré sur la figure 3.3, cette imbrication est utilisée comme la fonction de calcul d'un autre squelette (le squelette 1), alors ce dernier sera son squelette englobant ; elle deviendra du même coup une sous-imbrication de ce dernier. Or si l'imbrication du squelette 2 (englobant de la sous-imbrication) est licite avec le squelette 1 (le plus englobant de l'ensemble), alors les différents niveaux de sous-imbrications sont licites récursivement. Cela est vrai puisque aucun comportement de squelette n'a été modifié dans la chaîne d'imbrication.

C'est ici une relation de *transitivité* de l'imbrication de formes licites.

Si l'analyse des combinaisons licites est importante pour établir celles qui ont un sens et ainsi ne proposer que celles-ci aux utilisateurs, elle est aussi essentielle dans le développement de l'environnement. En effet elle permet de contrôler la construction du graphe au niveau du *front-end* de l'environnement et ainsi éviter que des combinaisons non exécutables soient prises en charge par l'environnement à l'exécution [6]. En fait nous verrons au chapitre 4 sur SKiPPER-II que la nouvelle version du noyau de SKiPPER est capable de s'accomoder de formes illicites de combinaisons en ce sens que le modèle d'exécution proposé est suffisamment *robuste* pour supporter l'exécution de telles combinaisons et garantir l'obtention d'un résultat (même si les performances obtenues sont alors médiocres).

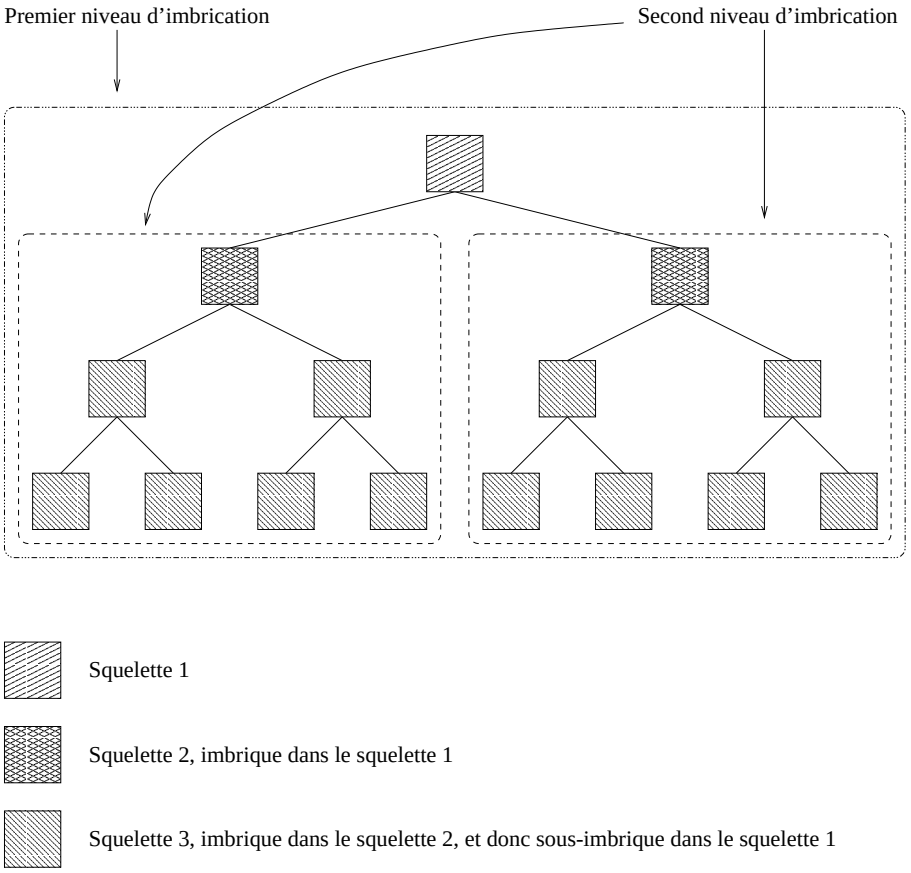


FIG. 3.3 – *Imbrication licite de squelettes.*

3.7.1 Imbrication dans un SCM

Les caractéristiques les plus importantes d'un squelette SCM pour décider des imbrications potentielles en son sein sont :

1. une même fonction de calcul doit occuper tous les processus dédiés aux traitements,

2. toutes ces fonctions doivent avoir approximativement le même temps d'exécution (pour garantir un bon équilibre de charge, et donc une bonne efficacité).

6. En plus d'offrir la possibilit´e d'informer l'utilisateur tr´es tˆot de l'exactitude de ses sch´emas de parall´elisation.

Le premier point signifie que l'imbrication n'a de sens que si les squelettes imbriqués sont tous les mêmes, c'est-à-dire que si on décide d'associer un SCM instancié à une fonction de calcul du SCM englobant, alors ce même SCM doit s'appliquer à toutes les autres fonctions de calcul sans exception.

Il est donc impossible d'utiliser des types de squelettes différents pour chaque fonction de calcul du SCM englobant, ainsi que d'utiliser un même type de squelette (par exemple un SCM) instancié différemment, *i.e.* dont les fonctions de calcul sont différentes de celles des autres squelettes.

La propriété 1 est une propriété *intrinsèque* du squelette SCM : elle fait partie de sa définition même. Elle est clairement exprimée dans sa sémantique déclarative :

```
let scm split compute merge x
        = merge ( map compute ( split x ) )
```

La même fonction (`compute`) est appliquée à toutes les données résultant du découpage de `x` par la fonction `split` (définition de `map`).

La propriété 2 traduit le fait qu'imbriquer dans un SCM un type de squelette dont le temps d'exécution varierait trop avec les données qu'il traite [7], ne serait pas judicieux (car conduirait à une mauvaise efficacité du schéma de parallélisation).

Cette propriété, le programmeur est libre de la respecter ou non. Si le temps de calcul de la fonction `compute` d'un SCM varie trop d'une donnée à l'autre, alors seule l'efficacité du squelette s'en trouve influencée, et décroît. En effet, rien dans l'expression syntaxique d'une telle imbrication ne permet de rejeter ce cas, comme dans l'exemple ci-dessous où un squelette DF est imbriqué dans une squelette SCM. Le programmeur doit juste s'assurer que le temps de traitement de chaque DF reste pratiquement constant.

```
let inner x = df compute acc z ( split_df x )
let outer x = scm split_scm inner merge x
```

avec :

```
split_df  : 'a      -> 'a list
split_scm : 'a      -> 'a list
compute   : 'a      -> 'b
acc       : 'c      -> 'b      -> 'c
merge     : 'c list -> 'd
z         : 'c
```

Les possibilités d'imbrications dans un SCM sont donc résumées dans le tableau ci-dessous :

SCM	oui
DF	déconseillée
TF	déconseillée

TAB. 3.1 – *Squelettes imbricables dans un SCM.*

7. Les donn´ees d'entr´ee de ce SCM imbriqu´e sont les r´esultats de l'application de la fonction *split* du SCM englobant.

3.7.2 Imbrication dans un DF et dans un TF

Nous traitons ici les cas du DF et du TF comme un seul puisque les caractéristiques les plus importantes de ces squelettes à prendre en compte leurs sont communes :

1. une même fonction de calcul doit occuper tous les processus dédiés aux traitements,
2. ces fonctions peuvent avoir des temps d'exécution différents.

Le premier point est exactement le même que pour un SCM (cf. section précédente). Il est dû aux choix faits lors de la conception de la base. Comme pour le SCM, il est donc impossible d'utiliser des types de squelettes différents pour chaque fonction de calcul du squelette englobant, mais aussi d'utiliser un même type de squelette (par exemple un SCM) instancié différemment, *i.e.* dont les fonctions de calcul sont différentes de celles des autres squelettes.

Contrairement au SCM, le second point autorise l'usage de squelettes dont les temps d'exécution peuvent dépendre des données qu'ils traitent. Il autorise donc ainsi l'utilisation d'un plus large spectre de squelettes pour l'imbrication.

Les possibilités d'imbrications sont résumées dans le tableau ci-dessous :

SCM	oui
DF	oui
TF	oui

TAB. 3.2 – *Squelettes imbricables dans un DF et dans un TF.*

3.8 Conclusion

Ce chapitre a présenté un tour d'horizon des environnements de programmation parallèle utilisant les squelettes algorithmiques et autorisant leur imbrication. Même si cette présentation n'est pas exhaustive elle met en avant les principaux projets développés à l'heure actuelle et montre simultannement différentes façon d'aborder le même problème.

Les environnements autorisant l'imbrication de squelettes ne sont pas nombreux. Cela tient pour l'essentiel à deux facteurs :

- le premier est bien entendu la question que l'on peut se poser quand à la réelle nécessite de permettre un tel niveau de composition pour des applications réelles ;
- le second est la difficulté de mise en œuvre d'une telle possibilité.

L'imbrication n'est pas une option triviale de la composition de squelettes. Elle pose des problèmes à plusieurs niveaux. Tout d'abord dans l'identification de l'imbrications de squelettes. Certains environnements choisissent une reconnaissance automatique dans la spécification du programme. La difficulté est ici l'automatisation de cette tâche. Dans d'autres, l'identification est explicite et s'appuie sur une aide fournie par l'utilisateur au moment de la spécification de son application. Dans ce cas, le travail de l'environnment est facilité, mais l'écriture du programme est alourdie.

L'autre niveau de difficulté est l'implémentation même de l'imbrication une fois identifiée. A ce moment-là plusieurs choix s'offrent au développeur de l'environnement de programmation : essayer de transformer le graphe de squelettes pour obtenir le même résultat sans imbrication (travaux de Darlington), traiter l'imbrication au moment de la compilation et donc tenter de résoudre les problèmes combinatoires associés (approche statique, HOPP, travaux de Michaelson, P^3L), ou enfin, traiter le problème à la volée lors de l'exécution en essayant d'optimiser localement la gestion des ressources disponibles (approche dynamique, EKTRAN). Chaque approche facilite une partie du travail mais reporte la difficulté sur l'autre. Chacune d'elle peut s'avérer plus efficace pour une classe d'applications particulière, mais la diversité des approches et l'absence (pour le moment) de solution simple et efficace a conduit à limiter le recours à l'imbrication dans les environnements de programmation parallèle.

C'est dans ce contexte que se place le développement de SKiPPER-II.

Chapitre 4

SKiPPER-II : une solution à l'imbrication de squelettes

SKiPPER-II est une nouvelle version de l'environnement d'aide à la programmation parallèle SKiPPER. Cette version a été développée pour autoriser la composition de squelettes algorithmiques, et tout spécialement l'imbrication, afin d'augmenter les capacités de parallélisation potentielle des applications de vision artificielle. Le noyau de l'environnement a été complétement réécrit pour supporter les nouveaux concepts utilisés, et plus particulièrement la mise en œuvre au moment de l'exécution d'un seul et unique modèle de squelette. Ce chapitre détaille les objectifs et les caractéristiques de SKiPPER-II.

4.1 Présentation

SKiPPER-II est une nouvelle version de l'environnement d'aide à la programmation parallèle SKiPPER et fait suite à la version SKiPPER-I présentée brièvement au chapitre 2 (le lecteur pourra se repporter à la thèse de D. Ginhac [Gin99] pour plus de détails). Il en reprend les fonctionnalités et les étend.

Son développement a été conduit par la volonté de proposer à l'utilisateur la possibilité de combiner de manière arbitraire plusieurs squelettes, là où la version précédente ne permettait que le séquencement. Pour ce faire, plusieurs étapes ont été nécessaires. En premier lieu, l'approche essentiellement statique présidant au fonctionnement de SKiPPER-I a été abandonnée face à la difficulté de proposer un modèle satisfaisant d'implantation statique des squelettes dynamiques (DF et TF). Un modèle d'exécution complètement dynamique a donc été élaboré. Ensuite, afin de rendre la composition des squelettes plus facile et régulière, un modèle générique de squelette a été construit. Ce modèle, indépendant de l'architecture cible, permet de donner une représentation unique à tous les squelettes de la bibliothèque. Il réduit ainsi le nombre de combinaisons envisagées lors de la composition des squelettes. De même, dans le cadre d'un modèle d'exécution complètement dynamique, il permet d'homogénéiser la représentation interne des squelettes.

4.2 Un modèle générique des squelettes

4.2.1 Le squelette TF/II

L'environnement de programmation SKiPPER propose à l'utilisateur quatre squelettes algorithmiques différents pour couvrir ses besoins en schémas de parallélisation dans le cadre d'applications de vision artificielle (cf. section 2.2.2). Parmi ces squelettes, les squelettes DF (Data Farming) et TF (Task Farming) reposent sur un modèle d'exécution dynamique, alors que celui des autres est essentiellement statique. Par dynamique on entend ici que les échanges de données et éventuellement le placement ou l'ordonnancement des processus le cas échéant ne peuvent être complètement calculés à la compilation. Certaines décisions doivent être prises «à la volée» lors de l'exécution. A l'opposé, pour un squelette «statique» les communications, le placement et l'ordonnancement des processus sont entièrement connus au moment de la compilation, et restent figés lors de l'exécution.

SKiPPER-I utilisait une représentation intermédiaire des squelettes sous la forme d'un Graphe Flot de Données Conditionné (GFDC) qui rendait difficile l'exploitation des squelettes «dynamiques» [1]. En effet, cela a conduit le développement de SKiPPER-I vers la mise en place d'une représentation intermédiaire «mixte» dans le sens où une partie du fonctionnement de ces squelettes (processus de gestion des communications dynamiques), à été «encapsulée» (cachée) au sein des fonctions de certains nœuds du graphe. L'inconvénient de cette démarche est que les schémas de parallélisation correspondant ne peuvent plus alors être manipulés indépendamment de leur implémentation.

La composition de squelettes étant un problème difficile (cf. chapitre 3), il est préférable de faire en sorte de le simplifier au plus tôt dans l'enchaînement des phases de développement d'une application avec l'environnement. Or l'approche développée dans SKiPPER-I conduit à une non-homogénéité dans les représentations intermédiaires des différents squelettes qui sont

1. Voir la section 2.3.6 page 68.

manipulées en interne par l'environnement. Cela induit une source de difficultés non négligeable lors de la mise en œuvre de la composition des squelettes car cette dernière ne peut alors pas être traitée complètement à un même niveau d'abstraction. Qui plus est, conserver plusieurs types de squelettes au niveau de la représentation intermédiaire donne une combinatoire importante pour les différents modes de composition.

SKiPPER-II a donc été conçu dans le but de n'avoir à manipuler qu'une seule et unique représentation interne pour tous les squelettes. Plus précisemment, même si quatre squelettes continuent à être proposés à l'utilisateur pour former son application, ils ne sont plus différenciés au niveau de leur représentation interne. Toute instance d'un squelette choisi par l'utilisateur est préalablement traduite en un squelette particulier avant toute opération dans la chaîne de traitement de l'environnement. Ce dernier squelette est un squelette générique qui peut assumer toutes les fonctions de tous les squelettes de la bibliothèque qui sont mis à la disposition de l'utilisateur (un «méta-squelette»). Ce squelette générique a été nommé TF/II pour Task Farming/version II. Les différents types de squelettes proposés à l'utilisateur ne sont plus que la traduction d'un certain type d'utilisation ou de comportement, par spécialisation, de ce «méta-squelette».

A partir de là, l'application de l'utilisateur qui peut être représentée sous la forme d'un graphe de squelettes est traduite sous la forme d'un graphe de TF/II avant toute manipulation dans le reste de l'environnement. L'intérêt est tout d'abord de réduire considérablement la combinatoire dans le cadre de la composition arbitraire de squelettes. A partir du graphe de TF/II, seules peuvent subsiter les cas du séquencement de deux TF/II, de leur mise en parallèle et de leur imbrication. Ainsi l'imbrication de deux squelettes pris arbitrairement dans la bibliothèque se réduit à la simple imbrication *systématique* de deux TF/II. Il n'y a donc plus plusieurs cas de figure à envisager, mais seulement un seul. La mise en œuvre de la composition s'en trouve donc facilitée.

En d'autres termes, SKiPPER-II ne manipule plus qu'une seule et unique représentation interne des squelettes. Dans le cadre d'un modèle exécutif complètement dynamique (ce sur quoi nous reviendrons), cette représentation interne de tous les squelettes est alors complètement indépendante de l'architecture cible.

Notons un autre atout de l'utilisation du squelette générique TF/II. Le schéma qu'il encapsule induit que le graphe proposé par l'utilisateur, s'il est licite, sera *toujours* transformé en un *arbre* par l'intermédiaire de la mise en œuvre de TF/II (cet aspect est décrit en détail à la section 4.2.2). En effet, chaque TF/II peut se représenter comme un arbre avec pour racine le processus maître et pour feuilles les processus esclaves. Les combinaisons de ces arbres sont toujours des arbres. Cela n'était pas le cas avec l'utilisation directe des squelettes de la bibliothèque de SKiPPER puisque le SCM ne se présente pas sous la forme d'un arbre. De plus, les combinaisons de squelettes autorisées ne permettent que de construire des graphes de squelettes sous la forme d'arbre. De ce fait, tout le graphe résultant de l'utilisation de TF/II est homogène à une structure arborescente, quel que soit le niveau d'observation : squelettes ou processus maîtres/esclaves. Cette structure offre un certain nombre d'avantages comme par exemple le fait qu'il n'y aura pas de cycles dans le graphe [2] (hormis, bien entendu, celui induit par l'utilisation d'un squelette

2. Les cycles peuvent s'av´erer d´elicats `a g´erer, notamment sur une imbrication profonde car la partie du graphe concern´ee par le cycle ne suit plus le d´eroulement temporel du reste du graphe, et il semble n´ecessaire de garantir dans ce cas que les ressources utilis´ees pour les squelettes impliqu´es dans le cycle leurs soient r´eserv´ees d'une it´eration sur l'autre, ce qui, dans l'´etat actuel du noyau, ne correspond pas `a la philosophie de l'approche, et ne peut donc ˆetre garanti.

ITERMEM, mais ce squelette apparaît toujours au niveau le plus englobant), ou bien encore des opportunités d'optmisation *a priori* du graphe afin de réduire le nombre de squelettes «actifs» au moment de l'exécution.

L'utilisation d'un unique squelette, et donc d'un seul modèle d'exécution, rend la partie opérationnelle (dorsal) de l'environnement plus facile à maintenir et à adapter. L'implémentation est alors limitée à la mise en œuvre du code d'un seul squelette. Toute modification ou évolution est plus aisément réalisable car il n'y a pas plusieurs squelettes à maintenir.

De la même façon, cette approche facilite l'introduction potentielle de nouveaux squelettes dans la base de SKiPPER : il suffit de définir une traduction du nouveau squelette ajouté (*via* sa représentation opérationnelle) sous forme d'un TF/II (cf. la section 4.5 page 111). Cette possibilité est l'un des atouts de cette version de SKiPPER, même si avec elle SKiPPER ne rentre pas encore dans le cercle très fermé des environnements de programmation parallèle fondés sur les squelettes qui autorise la déclaration de nouveaux squelettes par l'utilisateur lui-même (comme par exemple [GSP98]). Cette dernière possibilité peut cependant être envisagée si les développements futurs de cet environnement le demandaient car, aucune portion du code du noyau n'ayant à être générée ou modifiée en fonction du squelette, seule la manière d'affecter les fonctions utilisateurs aux différentes parties exécutables du squelettes TF/II doit être réalisé de manière automatique, ce qui peut être décrit par une formulation adéquate des squelettes de la base.

4.2.2 Transformation d'un graphe de squelettes en un arbre de TF/II

La représentation intermédiaire du squelette TF/II est très importante pour le fonctionnement de SKiPPER-II. En effet, c'est elle qui sera interprétée à l'exécution et qui dictera au noyau comment reproduire le comportement de tel ou tel squelette. Dès le début l'environnement ne manipule qu'un arbre de squelettes TF/II.

Le squelette TF/II peut être vu comme un squelette TF généralisé. Comme lui, il possède un processus maître et des processus esclaves et il adopte exactement le même comportement (cf. section 2.2.2.3 page 61). La seule différence est qu'il peut supporter des fonctions *différentes* pour chacun de ses processus esclaves.

Les figures 4.1 à 4.3 illustrent la transformation des squelettes de SKiPPER en TF/II. Considérons par exemple le cas de la transformation d'un squelette SCM en TF/II. Les cases blanches de la figure représentent des fonctions séquentielles (fournies par l'utilisateur), et les cases grises des processus du noyau, paramétrés par les fonctions séquentielles précédentes, mis en œuvre pour supporter la conversion de forme du squelette. On constate que les fonctions esclaves restent totalement inchangées, seul le maître du squelette TF/II doit être «synthétisé» à partir des fonctions SPLIT et MERGE du squelette SCM. En fait, il n'y a aucune synthèse de code pour le passage d'un squelette à un autre. Le processus maître est déjà complètement écrit dans le noyau de SKiPPER-II. Simplement, pour l'exécution, il sera paramétré par le nom des fonctions de l'utilisateur qui remplissent le rôle des fonctions SPLIT et MERGE du squelette SCM.

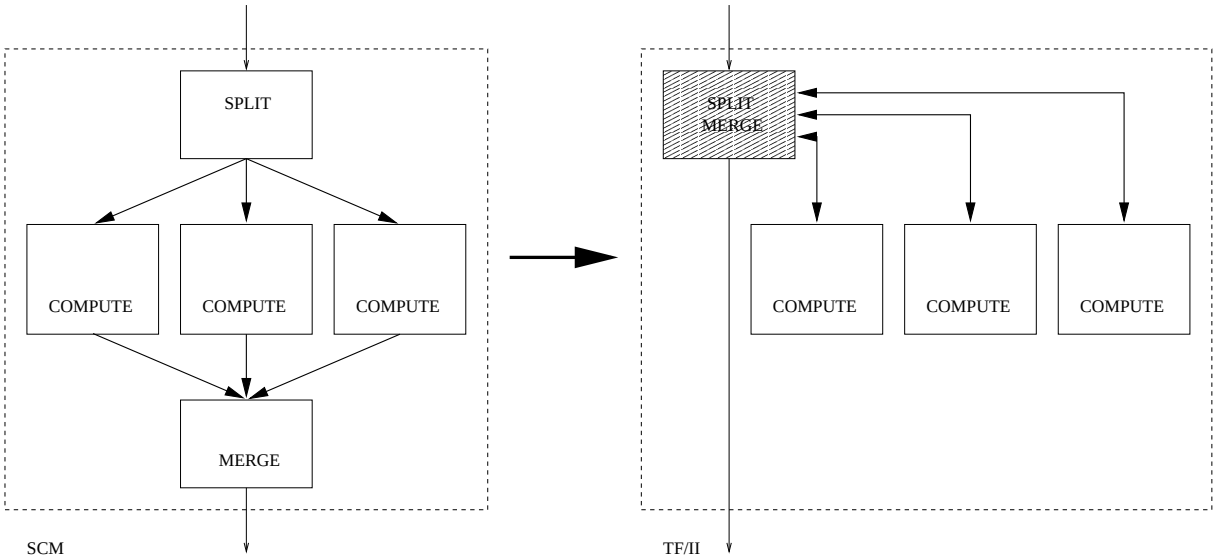


FIG. 4.1 – *Transformation d'un squelette SCM en squelette TF/II.*

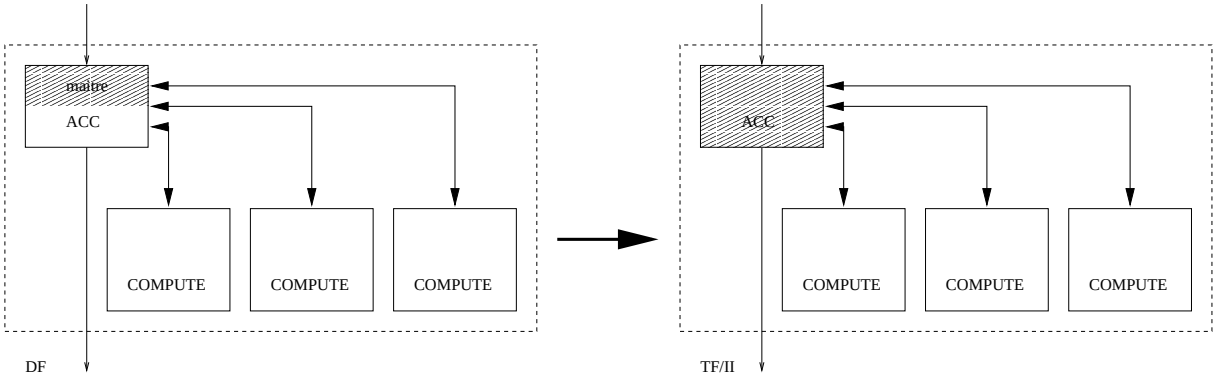


FIG. 4.2 – *Transformation d'un squelette DF en squelette TF/II.*

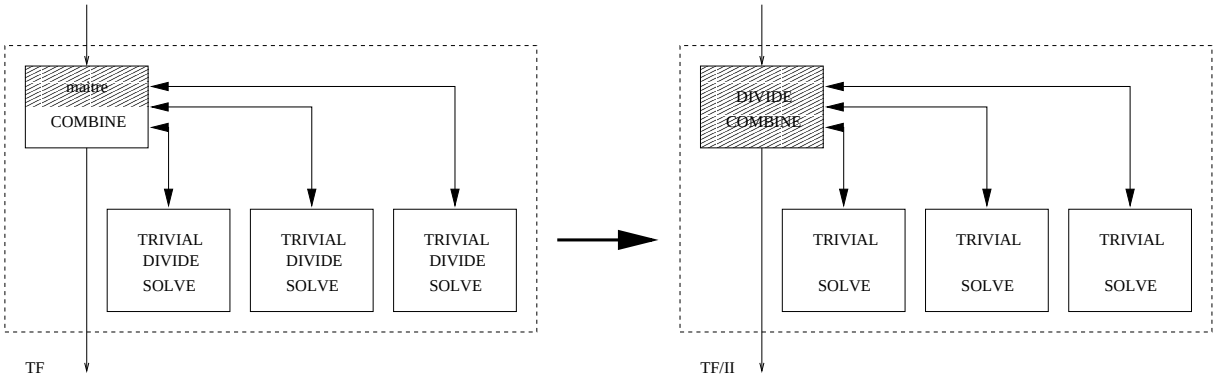


FIG. 4.3 – *Transformation d'un squelette TF en squelette TF/II.*

Ainsi, au moment de l'exécution, l'instance du processus maître du TF/II ne fait qu'appeler au moment opportun l'une où l'autre de ces deux fonctions utilisateur (voir la figure 4.4 page 95). C'est le seul point qui différencie une instance d'un processus maître d'une autre instance. Le code n'a donc pas à être généré au moment de la compilation.

La phase de compilation réalise donc la conversion de tous les squelettes [3] en une structure arborescente où chaque nœud correspond à une instance de TF/II (cette structure est décrite en détail à la section 4.3.2).

La figure 4.5 illustre cette phase avec un programme formé de deux SCM imbriqués.

4.3 Mécanismes de fonctionnement

4.3.1 Modèle d'exécution

Comme indiqué en début de chapitre, le modèle d'exécution choisi pour SKiPPER-II est complètement dynamique : toutes les décisions concernant le placement et l'ordonnancement des communications et des processus sont prises au moment de l'exécution et non plus lors de la compilation comme c'était le cas avec SKiPPER-I. De plus, le code exécutable n'est plus directement généré par l'environnement : SKiPPER-II s'appuie sur un noyau qui contient toutes les routines et mécanismes permettant l'exécution de l'application de l'utilisateur. Ainsi, le programme définitif et exécutable sur la machine cible se compose :

- du noyau de SKiPPER-II (appelé K/II par la suite),
- des fonctions de calcul de l'utilisateur,
- de données formant la représentation intermédiaire de l'organisation de l'application.

Les squelettes ne possédent plus de «code exécutable» spécifique à proprement parler. Ils sont émulés par le noyau en fonction des besoins de l'application conformément aux informations contenus dans la représentation intermédiaire. Les squelettes de type TF/II sont exécutés en faisant appel aux fonctions de calcul de l'utilisateur et suivant l'organisation de l'application proposée par la représentation intermédiaire.

Dans ce modèle d'exécution, les squelettes (ceux proposés à l'utilisateur) sont vus comme des processus concurrents (ou plus exactement comme un ensemble de processus) mis en concurrence pour obtenir les ressources nécessaires à leur exécution. Le caractère dynamique de notre modèle offre aussi la possibilité de parler de ressource disponible à un instant donné, le nombre de ressources et leur localisation pouvant évoluer au cours du temps.

3. Cette conversion reste encore `a l'heure actuelle r´ealis´ee pour l'essentiel `a la main car, mˆeme si de premiers d´eveloppements d'outils charg´es de cette ´etape ont d´ej`a vu le jour [BE01], ils sont encore `a l'´etat de prototypes.

```
/* Decoupage initial des donnees */
SPLIT()

Tant que [    des donnees sont encore a traiter
          || des processus esclaves sont encore en cours d'execution ]
Faire

  /* Demarrage d'autant d'esclaves que possible */
  Tant que [ il existe une ressource libre ]
  Faire
    Reserver une ressource aupres du PLServer pour lui allouer un esclave
    Transmettre le type d'esclave a la ressource reservee : fonction de calcul ou squelette
    Transmettre la donnee a traiter a la ressource reservee
  Fait

  Si [ il existe encore des esclaves en cours d'execution ]
  Alors
    /* Attente le retour de resultat de la part d'un esclave */
    Reception du type de resultat de la part d'un esclave

    Si [ la donnee a pu etre traitee ]
    Alors
      Reception du resultat
      Liberation de la ressource occupee par l'esclave
    Sinon
      Liberation anticipee de la ressource occupee par l'esclave
      /* Redecoupage de la donnee initiale pour traitement ulterieur */
      DIVIDE()
    Fin

  Sinon
    /* Execution d'esclaves en mode sequentiel */
    Recherche du type d'esclave a executer

    Si [ l'esclave est une fonction de calcul ]
    Alors

      Si [ PREDICATE() est vrai ]
      /* La fonction PREDICATE() du TF/II reprend la fonction TRIVIAL() du TF */
      Alors
        /* Les donnees peuvent etre traitees */
        SLAVE()
      Sinon
        /* Les donnees doivent etre redecoupees */
        DIVIDE()
      Fin

    Sinon
      /* L'esclave est un squelette */
      Re-execution de cet algorithme
    Fin

  Fin

  Si [ la donnee retournee par un esclave a ete traitee ]
  Alors
    MERGE()
    /* Cette fonction du TF/II reprend les fontions MERGE(), ACC() et COMBINE() des autres */
  Sinon
    Rien
  Fin

Fait
```

FIG. 4.4 – *Algortihme simplifié du processus maître d'un TF/II*

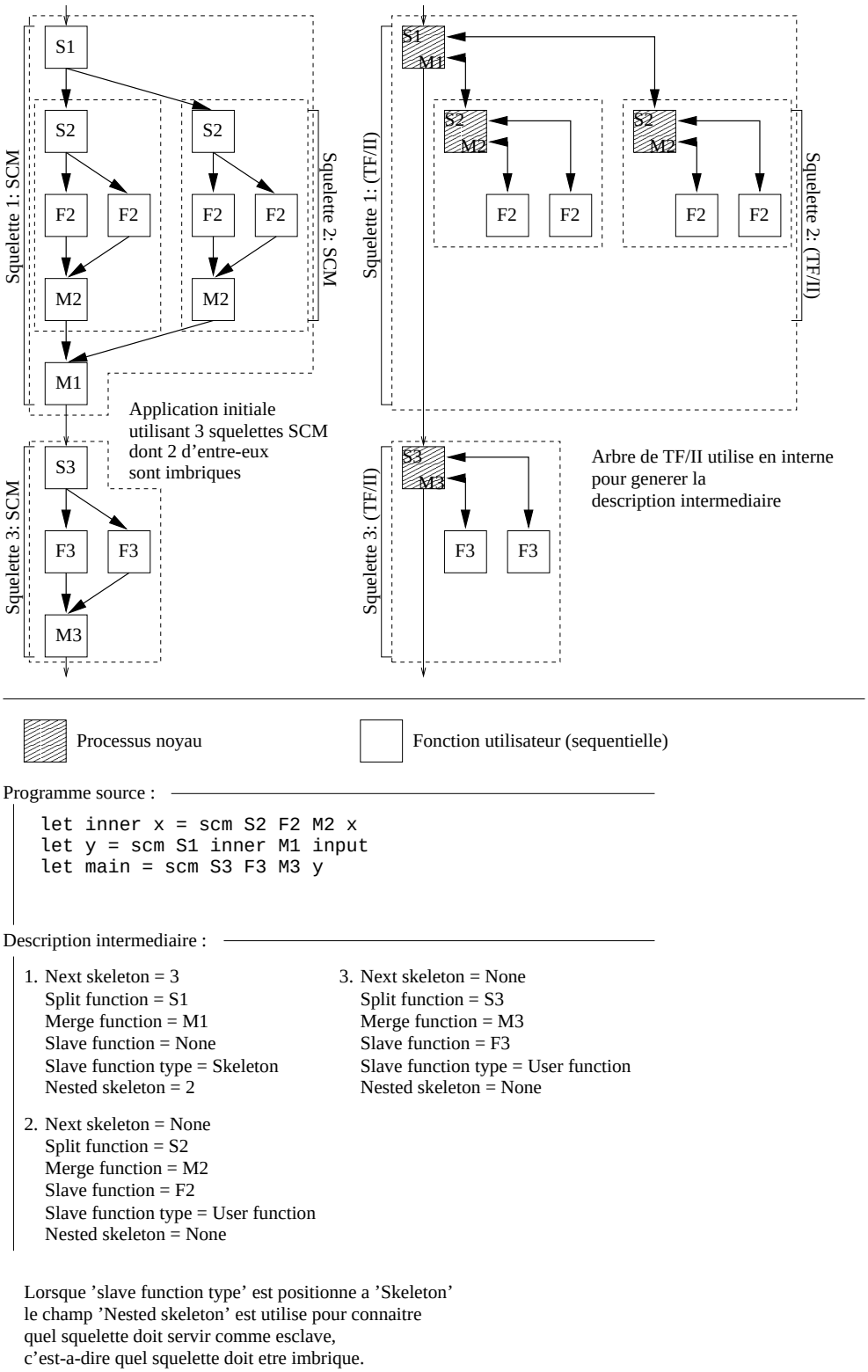


FIG. 4.5 – *Exemple de la structure de données utilisée pour la représentation intermédiaire.*

La source de parallélisme que nous utilisons est de type SPMD. Chaque processeur reçoit en conséquence une copie complète du code exécutable. En pratique, une ou plusieurs copies du noyau s'exécutent sur un processeur donné en fonction de ses capacités propres (de multitâche) et du niveau de performances souhaité par l'utilisateur. Lorsqu'un squelette TF/II doit être exécuté, une des copies du noyau se trouvant sur le réseau de processeurs de la machine cible joue le rôle du processus maître du TF/II. Cette copie gère alors tous les transferts de données entre le maître et les processus esclaves du TF/II.

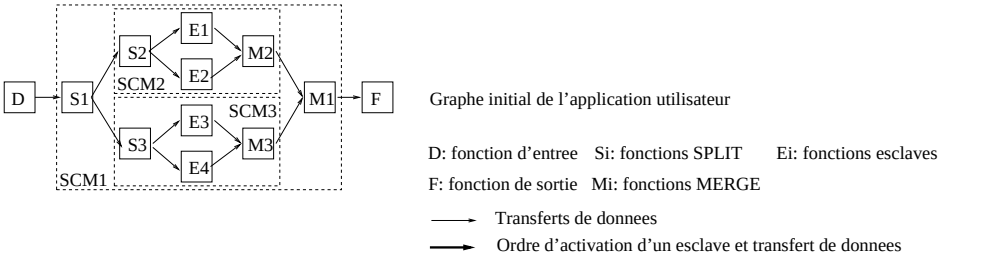
Une copie du noyau n'est pas dédiée à l'un des squelettes. Les copies réquisitionnées (temporairement) pour jouer le rôle du maître ou des esclaves d'un squelette TF/II le sont dynamiquement au moment de l'exécution en fonction des ressources disponibles. Les processus esclaves sont alloués par le processus maître. De cette manière, les copies du noyau interagissent pour émuler le comportement des squelettes. La gestion des ressources et l'interprétation de la représentation intermédiaire ne sont donc pas ici faites de manière centralisée, mais au contraire distribuée.

La figure 4.6 illustre sur un petit exemple le mécanisme décrit précédemment. L'exemple se compose de deux squelettes SCM imbriqués. On peut y voir le rôle que joue chaque copie du noyau (deux copies par processeur dans ce cas) lors de l'exécution.

Parce que chaque copie du noyau sait quand et où démarrer un nouveau squelette sans avoir à le demander aux autres copies, l'ordonnancement des squelettes peut être distribué. En réalité, chaque copie du noyau possède l'intégralité de la représentation intermédiaire. Cela implique que tout processeur peut démarrer un squelette lorsque c'est nécessaire en sachant simplement quel est le prédécesseur de ce squelette qui vient de terminer son exécution. Un nouveau squelette est alors lancé dès lors que son prédécesseur (dans l'arbre de TF/II) a pris fin. Le nouveau squelette est toujours démarré sur le même processeur que celui qui avait exécuté le processus maître du précédent (en effet, cette ressource est à coup sûr disponible et est la plus proche d'accès).

La mise en œuvre de l'imbrication de squelettes dans SKiPPER-II fait appel à une technique qui demande de communiquer peu de données supplémentaires par rapport aux données de l'utilisateur qui doivent circuler entre les squelettes de son application. Néanmoins, l'imbrication même de squelettes pose le délicat problème du temps passé à communiquer les données entre les différents «étages» de l'imbrication, et qui peut être un point critique en ce qui concerne les performances. En effet, l'imbrication suppose d'abord le transfert des données initiales à traiter vers le squelette le plus bas dans l'arbre des squelettes, et la récupération finalement des résultats de l'exécution par le squelette de niveau le plus élevé. Cette nécessité engendre normalement sur une machine à mémoire distribuée un surcoût en temps de communication du simple fait du transit des informations sur le réseau, non pas à des fins de traitement immédiat, mais de mise à disposition de ces données aux processus de traitement. La solution communemment avancée est de privilégier l'utilisation d'une machine à mémoire partagée.

Ce dernier type de machines posant des problèmes de contingeance avec un nombre élevé de processeurs, SKiPPER-II résout simplement ce problème en placant les processus maître de deux squelettes de niveaux différents ayant à communiquer entre eux sur la même ressource de calcul. Les communications se font alors via la mémoire vive du processeur, ce qui réduit la latence due à ce phénomène.



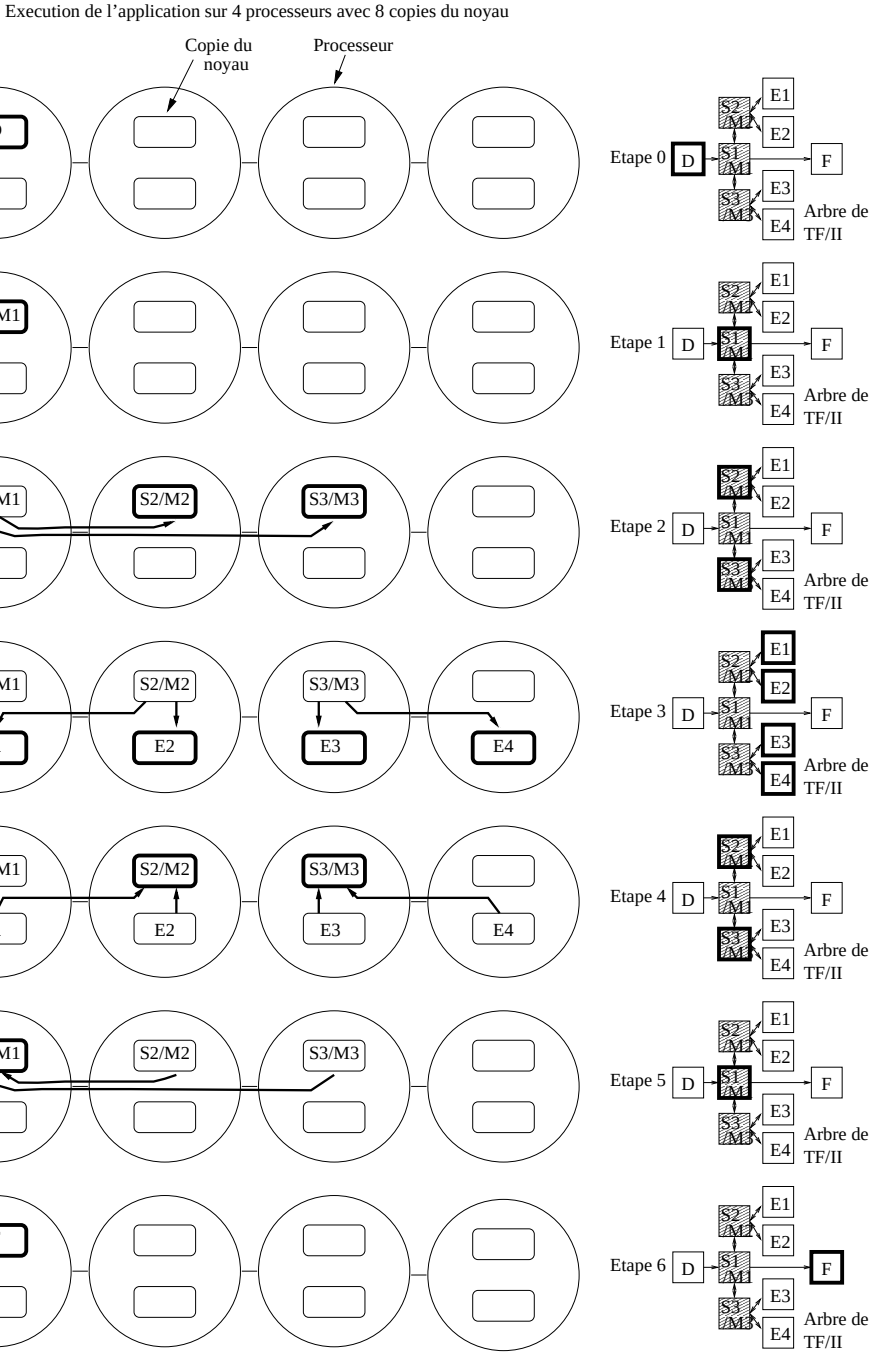


FIG. 4.6 – *Exemple d'exécution de deux SCM imbriqués.*

4.3.2 Repésentation intermédiaire de l'application

La représentation de l'application qu'utilise SKiPPER-II en interne (et que nous nommons *représentation intermédiaire*) est une description directement dérivée de la structure arborescente de TF/II. La structure de données employée pour la supporter a été choisie volontairement simple et fait appel à une structure de donnée native du langage C utilisé pour écrire le noyau de manière à limiter au maximum les besoins d'interprétation au moment de son traitement. En effet l'analyse de cette structure est faite tout au long de l'exécution de l'application : elle guide le noyau de SKiPPER-II dans les actions qu'il a à mener.

Un des intérêts de cette manière de procéder est que la représentation intermédiaire est ainsi compilée directement avec le code exécutable final et fait donc partir intégrante de ce dernier (cf. section 4.3.3).

La structure de données C utilisée pour encoder la représentation intermédiaire gérée par le K/II est un tableau. A chaque ligne de ce tableau correspond un squelette du graphe de l'application (un TF/II).

Le contenu de chaque ligne est le suivant [4] :

- la première colonne est l'identificateur du squelette,
- la seconde colonne indique si les résultats produits par le squelette doivent être communiqués au squelette qui le suit dans le graphe ou au squelette englobant en cas d'imbrication,
- la troisième colonne spécifie le statut du squelette en cas d'imbrication, ce qui permet de savoir si une fonction de calcul pure doit être exécutée pour ce squelette ou non,
- la dernière colonne spécifie l'identificateur du squelette à imbriquer dans le cas d'une imbrication.

La deuxième colonne joue un rôle crucial lors de l'imbrication de squelettes. En effet, en dehors de toute imbrication, les résultats des traitements obtenus par un squelette donnée sont passés au suivant dans le graphe de l'application. En cas d'imbrication les résultats obtenus par un squelette imbriqué doivent être communiqués au squelette englobant puisque le squelette interne fait office de fonction de calcul (*compute*) pour le squelette englobant. C'est ce dernier qui se chargera (éventuellement) de les communiquer à un autre squelette. Cette colonne sert aussi à indiquer si un squelette est le dernier squelette du graphe.

La figure 4.7 donne un exemple de la structure C encodée pour l'exemple d'application donnée en figure 4.5 page 96.

4. La description que nous faisons ici, si elle rend fid`element compte des capacit´es de la repr´esentation interm´ediaire, a ´et´e l´eg`erement mod´ifi´ee pour am´elior´ee la lisibilit´e.

```
#define SKL_NBR 3
SK2_Desc app_desc[SKL_NBR] =
{
   {SK1, NEXT      , MASTER, SK2 },
   {SK3, END_OF_APP, SLAVE , NIL },
   {SK2, UPPER     , SLAVE , NIL }
};
```

FIG. 4.7 – *Exemple de représentation intermédiaire*

Accompagnant cette structure de données (qui correspond à proprement parler à la description intermédiaire utilisée sous SKiPPER-II), un ensemble de tableaux C vient compléter le dispositif. Ces tableaux ont pour rôle de faire la liaison entre les squelettes et les fonctions utilisateur qu'ils doivent exécuter. Pour chacune des fonction d'un TF/II qui peut être instanciée par une fonction utilisateur (par exemple *split* et *compute*) existe un tableau. Chaque tableau a autant d'entrées qu'il y a de squelettes dans l'application. Une entrée est constituée d'un pointeur vers la fonction C à exécuter. Ici encore, la structure de donnée choisie permet d'être compilée directement dans le code exécutable et décodée rapidement par le noyau (ici sans aucune interprétation). L'exemple de la figure 4.8 montre l'utilisation de ces tableaux dans le cas de l'application donnée en figure 4.5 (les arguments des fonctions systèmes n'ont pas été reproduits pour simplifier l'écriture).

4.3.3 Génération du code exécutable et noyau de SKiPPER-II

Le code exécutable d'une application est constitué des éléments suivants (voir figure 4.9 page suivante pour une vue synthétique) :

– le noyau (K/II) de SKiPPER-II,

– les fonctions de l'utilisateur (fonctions séquentielles, en C ou C++),

– la représentation intermédiaire de l'organisation de l'application.

L'ensemble de ces éléments (ressources) est répliqué au moins une fois par unité de calcul (processeur physique). Un processeur peut même, s'il a des capacités de multitâches (soit de manière native comme avec le T9000 d'Inmos, soit grâce au système d'exploitation), en accueillir plusieurs copies. L'intérêt de disposer de plusieurs copies du code exécutable sur chaque processeur est de permettre comme nous allons le voir l'exécution simultanée de processus maître de squelettes et d'un processus esclave pour optimiser l'exploitation des ressources de calcul.

```
int (* SPLIT[SKL_NBR])()=
{
  &S1,
  &S2,
  &S3
};

int (* PREDICATE[SKL_NBR])()=
{
   UNDEFINED,
   STANDARD_PREDICATE,
   STANDARD_PREDICATE
};

int (* DIVIDE[SKL_NBR])()=
{
   UNDEFINED,
   STANDARD_DIVIDE,
   STANDARD_DIVIDE
};

int (* SLAVES[SKL_NBR])()=
{
   UNDEFINED,
  &F2,
  &F3
};

int (* MERGE[SKL_NBR])()=
{
  &M1,
  &M2,
  &M3
};
```

```
Notes : - la mention UNDEFINED pour une entree signifie qu'elle est
          sans objet,
        - la mention STANDARD_... signifie qu'une fonction speciale
          du noyau est utilisee en remplacement d'une fonction
          utilisateur.
```

FIG. 4.8 – *Exemple de points d'entrées de fonctions utilisateur*

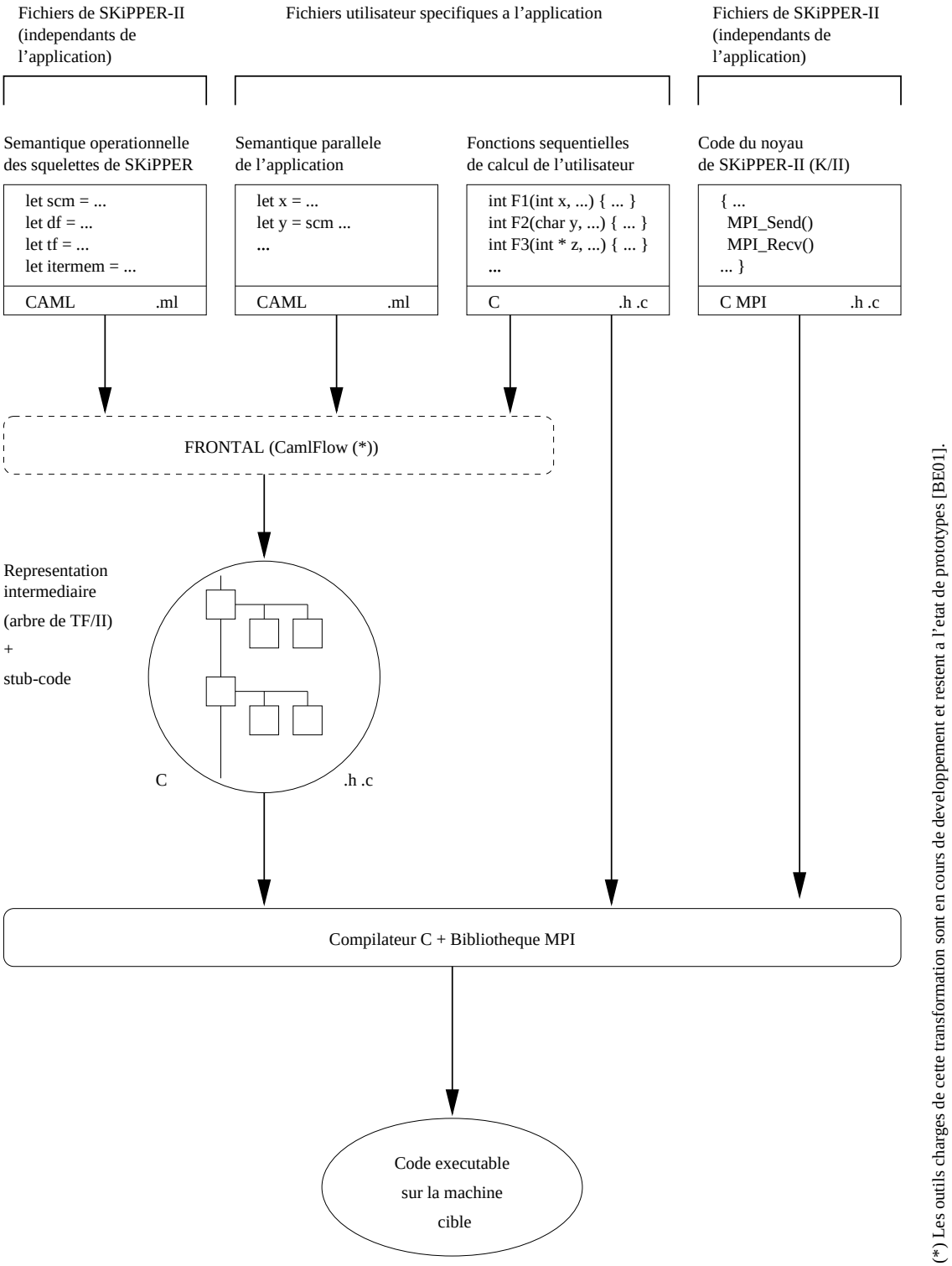


FIG. 4.9 – *Synoptique de SKiPPER-II.*

Le noyau de SKiPPER-II contient tout le logiciel nécessaire à l'exécution du squelette générique TF/II de manière distribuée. Lorsque les copies du noyau sont distribuées sur le réseau de processeurs, elles sont toutes placées dans un mode qui leur permet d'assumer indifféremment le rôle d'un processus maître ou d'un processus esclave du TF/II. Cela est valable en fait pour toutes les copies à l'exception de deux d'entre elles. En effet une des copies du noyau doit être désignée pour exécuter la fonction d'entrée du graphe de squelettes de l'application. Cette fonction d'entrée est identique à celle déjà existante dans les graphes d'applications utilisés dans la version SKiPPER-I. Son rôle est simplement d'acquérir les données en provenance des capteurs et à les transmettre à la chaîne de traitement, éventuellement après un premier traitement. La copie du noyau supportant l'exécution de cette fonction est aussi celle qui supportera l'exécution du processus maître du premier squelette. L'autre copie qui a un rôle particulier gardera cette fonction spéciale durant toute l'exécution de l'application. Elle ne participera jamais en tant que telle à l'exécution d'un quelconque squelette. Cette copie est un processus désigné sous le nom de PLServer. Son rôle est de gérer la disponibilité des ressources de calcul. Ainsi lorsqu'un processus maître veut déployer ses esclaves, il interroge le PLServer pour connaître les ressources disponibles et les réserver.

4.3.4 Transactions internes

Au cours de l'exécution d'une application, les copies du noyau qui ont été démarrées sur les différents processeurs dialoguent entre elles pour assurer le déroulement correct de l'application, c'est-à-dire conforme à la représentation intermédiaire.

Les échanges se font :

– avec le serveur de ressources,

– entre les copies du noyau.

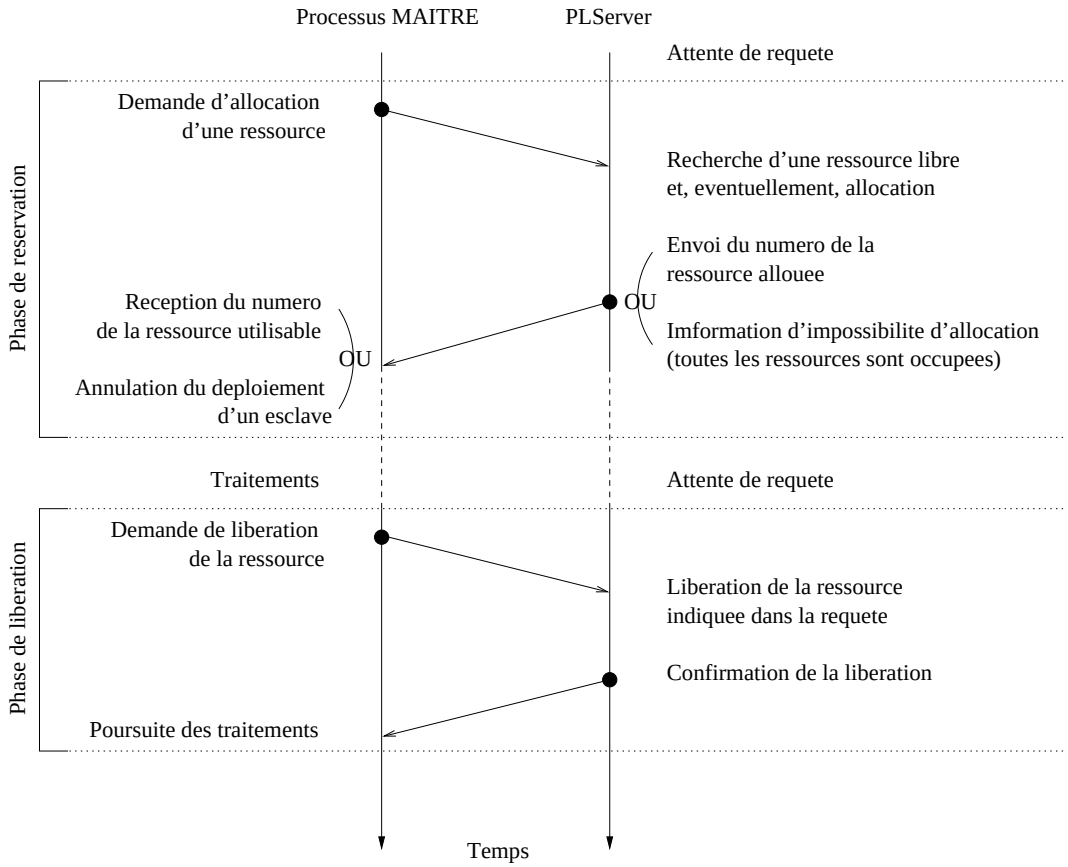
4.3.4.1 Avec le serveur de ressources

Le processus serveur de ressources, ou PLServer, est un processus destiné à tenir à jour une liste des ressources disponibles et non utilisées sur le réseau de la machine cible et à autoriser ou non leur accès. Une ressource pour le noyau de SKiPPER-II est une copie du noyau. En effet ce sont ces copies qui vont permettre le déroulement d'une séquence de calcul. Il faut distinguer ce type de ressource logicielle des processeurs physiquement disponibles sur la machine cible : chaque processeur peut être chargé avec plusieurs copies du noyau [5].

Lors de l'exécution d'un squelette (unité exécutive de base), un certain nombre de copies du noyau vont coopérer pour émuler le fonctionnement d'un TF/II. Celle qui joue le rôle du processus maître est en charge du déploiement sur les ressources disponibles des esclaves du squelette. Cela signifie qu'elle doit trouver des ressources disponibles, les réserver, démarrer un processus esclave sur chacune d'elles et communiquer avec elles pour exécuter le squelette. Pour chaque esclave à activer le processus maître interroge le PLServer pour lui demander l'allocation d'une ressource. Le PLServer répond alors avec l'identifiant de la ressource disponible qu'il va réserver exclusivement à ce squelette (cf. figure 4.10). A partir de cet instant les communications entre le maître et la copie du noyau désignée comme esclave pourra se faire de

5. Le nombre de ressources disponibles ne sera ´equivalent au nombre de processeurs disponibles que si une unique copie du noyau est charg´ee sur chaque processeur.

manière sécurisée étant donné qu'aucun autre processus ne pourra accéder à cette ressource. Si aucune ressource n'est disponible pour l'exécution de l'esclave, le PLServer en informe le processus maître qui pourra décider de poursuivre l'exécution du squelette malgré la pénurie de ressources, notamment en en séquentialisant une partie. Lorsque l'exécution d'un esclave est terminée, le maître signale au PLServer. Ce dernier met alors à jour ses tables pour rendre la ressource libérée disponible pour d'autres squelettes. Pour autoriser l'attribution de certaines ressources à un squelette, le PLServer tient compte de la position des copies du noyau qui sont libres (voir la figure 4.11 qui donne l'algorithme simplifié du PLServer). Mais l'allocation de ressources se fait aussi en fonction de la nature du processus qui doit être implanté et exécuté. Sur un même processeur, seul un unique esclave peut être implanté alors que plusieurs processus maître sont autorisés à s'exécuter simultanément. Ainsi, si un processeur dispose de plusieurs copies du noyau, à un instant donné seule une unique copie de cet ensemble peut être utilisée comme esclave alors que plusieurs autres peuvent être utilisées comme maître. La raison de ce mécanisme est la suivante. Les processus maître font très peu de calculs contrairement aux processus esclaves et passent la majorité de leur temps à attendre des données, ils peuvent être exécutés en pseudo-parallélisme sur un unique processeur (sans pour autant gêner l'exécution d'un esclave). Ainsi placer plusieurs maîtres sur un même processeur optimise l'occupation des unités de calcul. Par contre placer plusieurs esclaves sur un même processeur reviendrait à faire chuter les performances de l'application car le parallélisme entre ces esclaves ne serait pas réel.



Dans le cas ou la ressource ne peut etre allouee par le PLServer, seule la phase de reservation du diagramme entre en jeu, aucune ressource n'ayant plus a etre liberee puisque non reservee.

FIG. 4.10 – *Protocole de communication entre le processus maître d'un TF/II et le PLServer.*

```
Tant que [ application non terminee ]
Faire

  Attendre une requete

  Selon [ requete ]
  Tester si

    = allouer une ressource :

      Si [ pour un esclave ]
      Alors

        Faire
          Rechercher une ressource libre dans la table des ressources
          Si [ le processeur de la ressource libre ne possede
               aucune autre ressource affectee a un esclave   ]
          Alors
            Mettre a jour la table des ressources
            Confirmer l'allocation au processus demandeur
            Arreter la recherche de ressources libres
          Sinon
            Continuer la recherche de ressources libres
          Fin
        Tant que [ une ressource libre n'a pas ete trouvee
                   et qu'il reste des ressources a evaluer ]

        Si [ aucune ressource n'est libre ]
        Alors
          Indiquer qu'aucune ressource n'est libre au processus demandeur
        Sinon
          Rien
        Fin

      Sinon

        /* Pour un processus maitre */
        Faire
          Rechercher une ressource libre dans la table des ressources
          Si [ une ressource libre a ete trouvee ]
          Alors
            Mettre a jour la table des ressources
            Confirmer l'allocation au processus demandeur
            Arreter la recherche de ressources libres
          Sinon
            Continuer la recherche de ressources libres
          Fin
        Tant que [ une ressource libre n'a pas ete trouvee
                   et qu'il reste des ressources a evaluer ]

        Si [ aucune ressource n'est libre ]
        Alors
          Indiquer qu'aucune ressource n'est libre au processus demandeur
        Sinon
          Rien
        Fin

      Fin

    = liberer une ressource :

      Mettre a jour la table des ressources
      Confirmer la liberation au processus demandeur

  Fin

Fait
```

FIG. 4.11 – *Algorithme simplifié du PLServer*

4.3.4.2 Entre les copies du noyau

Les transactions qui s'établissent entre les copies du noyau sont de deux natures :

– communication de données utilisateurs,

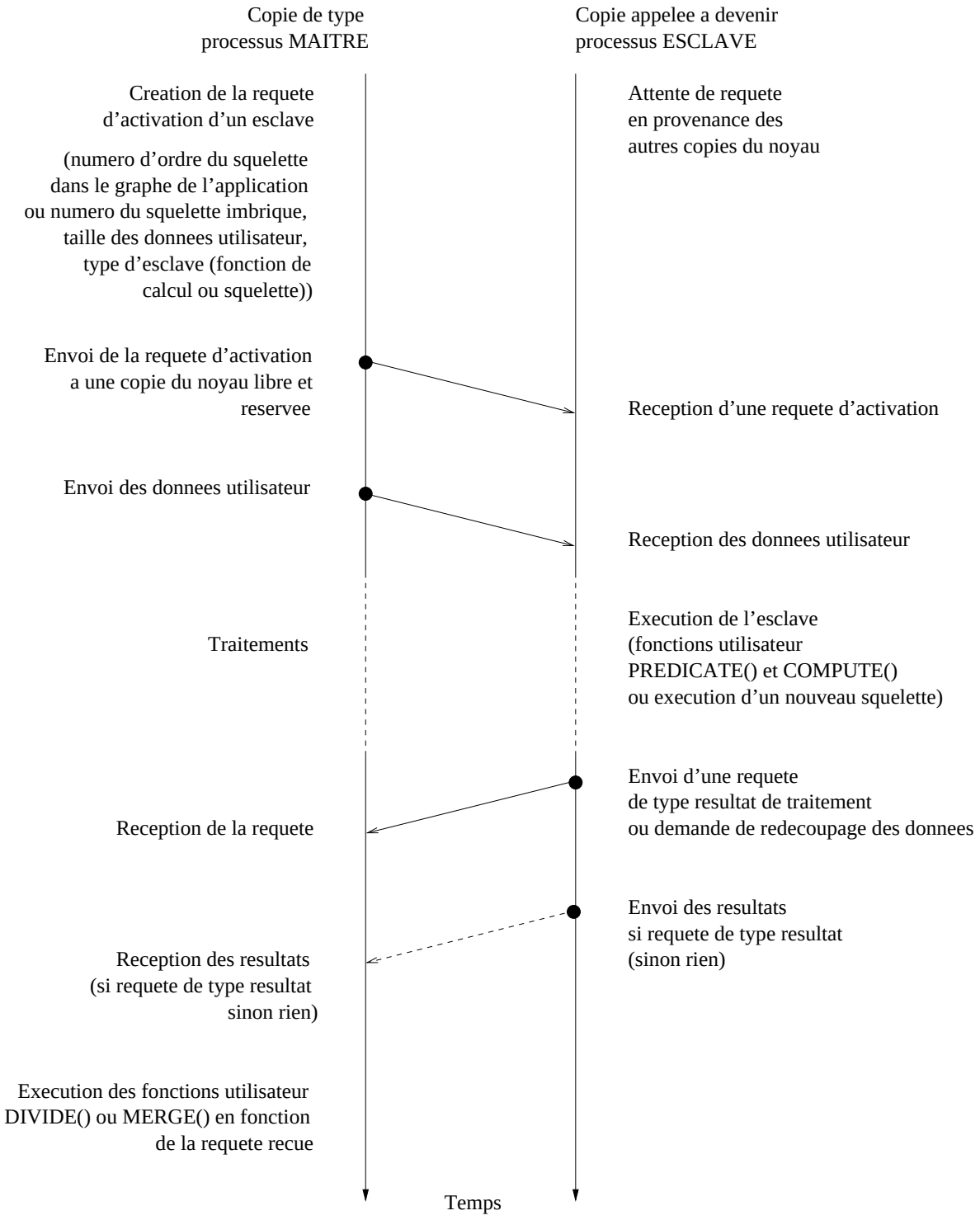
– communication d'ordres et de données pour le système.

La première catégorie de transactions concerne les données que les fonctions utilisateur doivent s'échanger dans le cadre du déroulement normal de l'application. Ce sont donc les données d'entrée de l'algorithme, et toutes les données intermédiaires produites par les différentes étapes des traitments.

La seconde catégorie concerne quant à elle les échanges de nature à assurer la distribution et l'ordonnancement correct des fonctions de l'application et des processus du noyau. Tous ces échanges se font exclusivement entre fonctions systèmes du noyau et entre des copies qui sont enrolées dans l'exécution d'un squelette. Ces transactions permettent de donner l'ordre à une copie de devenir un processus esclave au sein d'un squelette (cf. figure 4.12).

Le maître, une fois qu'il a réservé une ressource pour l'un de ses esclaves, adresse à la copie du noyau concernée une requête lui indiquant qu'elle devient un processus esclave. La copie cible du noyau acquiert à ce moment les informations lui permettant de mettre en œuvre cet ordre et notamment de quel squelette elle fait partie. Elle peut ainsi aller lire dans la représentation intermédiaire de l'application qui se trouve dans sa mémoire, quelle fonction utilisateur elle doit exécuter. Il se peut aussi que l'esclave corresponde en fait à un autre squelette imbriqué dans le premier. Dans ce cas c'est un processus maître qui est démarré en lieu et place du processus esclave. Les données propres à l'application complètent alors ce type d'échange pour faire le lien entre les différents éléments du squelette une fois qu'il est rendu complètement opérationnel.

Grâce à l'échange simple de requêtes entre les différentes copies du noyau, et au fait que chaque copie dispose de toutes les fonctions utilisateurs et de la représentation intermédiaire, il n'y a pas à proprement parler de migration de processus lorsqu'il s'agit d'activer un esclave par exemple. En effet, aucun morceau de programme n'est transmis d'une copie à l'autre. Seul un numéro d'ordre est échangé. Cela permet de rendre l'activation d'un processus, maître ou esclave extrêmement rapide (au détriment de l'occupation mémoire).



Notes :
- la fonction PREDICATE() du TF/II reprend la fonction TRIVIAL() du TF,
- la fonction MERGE() du TF/II reprend les fonctions MERGE(), ACC() et COMBINE() des autres squelettes.

FIG. 4.12 – *Protocole de communication entre une copie maître du noyau et une copie appelée à devenir esclave.*

4.3.5 Cas du manque de ressources et émulation séquentielle

Lors du déploiement des esclaves d'un squelette par le processus maître, il peut arriver que, du fait notamment de la concurrence entre les squelettes, le déploiement ne puisse se faire complètement. Dans le cas d'une pénurie de ressources, le fonctionnement normal est d'utiliser au maximum les ressources disponibles sur le moment en attendant que d'autres se libèrent. En effet, à chaque fois qu'un esclave a fini son travail, le PLServer est interrogé non seulement pour réaffecter cette ressource, mais aussi pour vérifier s'il n'y aurait pas d'autres ressources nouvellement disponibles. Ainsi le squelette réadapte sa «géométrie» aux ressources disponibles tout au long de son exécution. Ce comportement est illustré sur le synoptique de la figure 4.13. Cet exemple montre l'exécution simultannée de deux TF/II sur un réseau de 6 ressources (copies du noyau). Le premier squelette nécessite l'exécution de deux esclaves (S1A et S1B), alors que le second en nécessite 5 (S2A à S2E). Considérant que les deux processus maîtres démarrent simultanément, ils occupent chacun une ressource (étape 1). A l'étape suivante ils déploient leurs esclaves. Dans le cas présenté sur la figure, le squelette 1 a réussi à déployer l'intégralité de ses esclaves en une seule fois, mais le second squelette, lui, a échoué. De ce fait, 3 de ses esclaves sont mis en attente jusqu'à ce que des ressources se libèrent. C'est ce qui se produit à la fin de cette étape 2 où les esclaves S1A et S1B ont terminé leur exécution après avoir retourné leurs résultats à leur maître M1 (d'où sa persistance sur la première ressource). Le second squelette (en supposant ici que ses deux premiers esclaves ont eux aussi terminés leur exécution à l'étape 2) peut alors déployer ses 3 derniers esclaves restés en suspend puisqu'il dispose à ce moment de 4 ressources libres (étape 3). L'étape 4 est la dernière phase de l'existence des squelettes, les deux maîtres n'ayant plus alors qu'à préparer les résultats en leur possession pour la transmission vers les squelettes en aval.

De ce fonctionnement découle la gestion du cas critique où aucune ressource n'est disponible au moment du déploiement des esclaves. Dans ce cas, le maître décide d'exécuter le premier esclave sur sa propre copie du noyau. Le maître est alors suspendu et le premier esclave s'exécute à sa place. Lorsqu'il a terminé, la main est rendue au maître qui exécute alors l'esclave suivant, soit de la même manière si la pénurie persiste, soit sur une autre copie du noyau si des ressources se sont libérées entre-temps.

Cette caractéristique dans le fonctionnement du noyau fait que l'émulation séquentielle de l'application, c'est-à-dire la possibilité d'exécuter séquentiellement un code spécifié de manière parallèle, n'est plus considérée comme un cas particulier dans SKiPPER-II. Cette manière d'exécuter l'application se produit naturellement à partir du moment où SKiPPER-II constate qu'il n'y a qu'une seule ressource disponible. Ainsi, et contrairement au fonctionnement de SKiPPER-I, il n'est plus nécessaire de fournir une définition séquentielle des squelettes pour obtenir une émulation séquentielle d'une application, définition séquentielle qui présentait deux inconvénients : le premier de risquer de ne pas correspondre exactement au comportement du squelette dans sa version parallèle, le second de nécessiter la mise à jour de cette version de chaque squelette à chaque modification de la version parallèle. Concernant l'intérêt de disposer d'une émulation séquentielle, il faut rappeler que certains environnements de développement attachés à certaines machines parallèles, telles que les machines prototypes dédiées, disposent rarement de tous les outils de débogage classiques. La recherche d'erreurs algorithmiques ou d'implantation est par conséquent souvent fastidieuse et difficile. L'émulation séquentielle permet alors de dissocier les phases de mise au point de l'algorithme proprement dit, et celle de validation opérationnelle, en conditions réelles, sur la machine cible [SGCD01].

On notera enfin que l'application n'a plus à être recompilée entre une exécution séquentielle

et une exécution parallèle puisque les deux cas ne se distinguent plus au niveau du programme, mais seulement au niveau du nombre de ressources affectées à cette application. De ce fait, même on s'affranchit de l'une des contraintes de la précédente version de SKiPPER qui était de devoir recompiler d'une manière différente l'intégralité de l'exécutable pour chaque expérimentation avec un nombre de processeurs différent. Ici, il suffit de relancer l'application avec plus ou moins de processeurs dans le réseau pour que SKiPPER-II prenne en compte ce changement dynamiquement, sans recompilation.

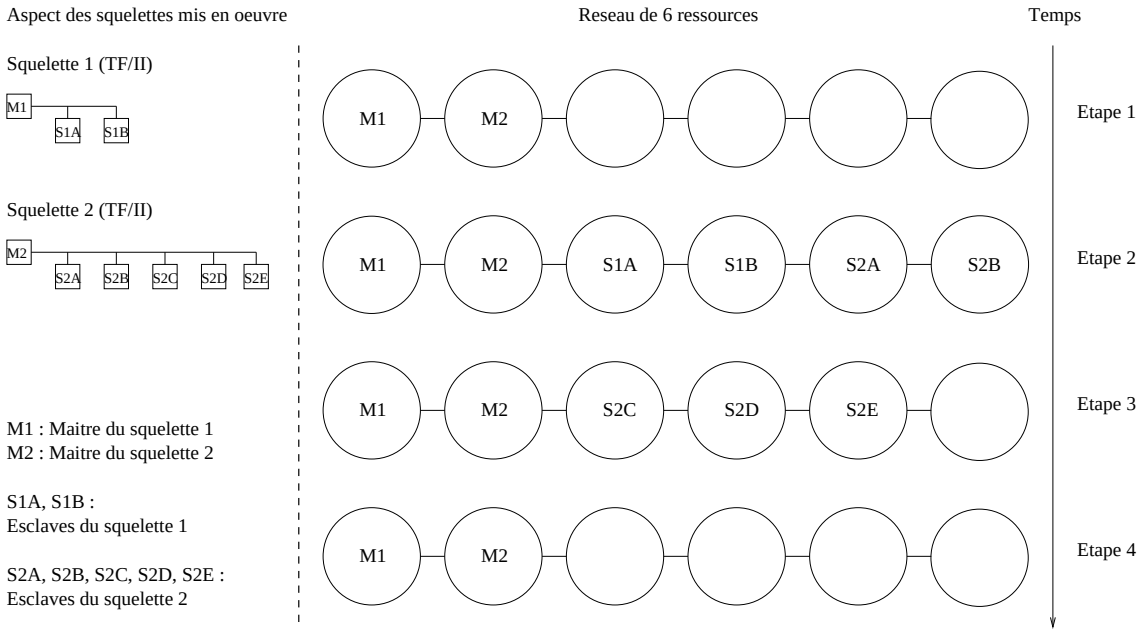


FIG. 4.13 – *Comportement en cas de manque de ressources.*

4.4 Portabilité du noyau

La prise en compte de la portabilité du nouveau noyau de SKiPPER a été considérée dès le début de son développement et est restée présente comme une nécessité importante tout au long de ce développement. Cette prise en compte précoce garantit au noyau une portabilité maximale.

La question de la portabilité est essentielle dans le cadre du développement d'un environnement devant aider les utilisateurs de machines parallèles à les programmer [ST98]. A ce titre la portabilité de l'environnement est une priorité car une application développée avec l'aide de cet environnement ne peut être portable si l'environnement lui-même ne l'est pas ! Cette propriété du noyau participe à la portabilité de l'application dans son ensemble qui est cruciale quand on sait les problèmes de réutilisabilité de code qui existent en programmation parallèle notamment du fait de la diversité des architectures et de leurs moyens de programmation. La difficulté de réutiliser, dans une nouvelle application, un certain nombre d'algorithmes développés pour une autre est en effet un frein sérieux à une réelle émergence de l'informatique parallèle dans beaucoup de domaines applicatifs [6].

6. La simple mise `a disposition de standards de communication comme MPI, PVM ou OpenMP ces derni`eres ann´ees a, `a elle seule, apport´e un grand changement dans l'acc`es `a la programmation parall`ele pour une com-

L'accent mis sur la portabilité du noyau de SKiPPER-II lui permet de s'accommoder de nombreuses plates-formes matérielles et ce de manière *immédiate* par simple recompilation, sans aucune ligne de code à modifier [7]. En effet, le noyau a été développé avec la volonté de ne pas faire appel à des logiciels ou bibliothèques tiers qui ne soient pas standards ou insuffisamment diffusés. C'est ainsi que le noyau K/II de SKiPPER peut être installé sur toute machine [8] disposant simplement d'un compilateur C standard (GCC par exemple) et de la bibliothèque de communication MPI. MPI étant largement diffusée pour une large gamme de machines, la seule nécessité de la présence de ces deux outils participe à la garantie d'une portabilité excellente de l'environnement. De plus, la bibliothèque MPI [GLS94], du fait de son interface standardisée, documentée et indépendante de toute implémentation, peut être facilement mise en œuvre sur des plates-formes spécifiques n'en disposant pas par défaut. Ce fut notamment le cas pour la machine TransAlpha [PAR96b] [PAR96a] conçue et développée au sein même de notre laboratoire [GT00], pour laquelle seul un petit sous-ensemble de MPI a dû faire l'objet d'un portage. En effet, un effort tout particulier a été fait sur l'utilisation de MPI par le noyau K/II puisque ce dernier n'utilise qu'un jeu extrêmement restreint des fonctions de communication disponibles dans le standard MPI : par exemple concernant les fonctions de communication proprement dites, seules les fonctions `MPI_SSend` et `MPI_Recv`, qui sont les fonctions de communication les plus élémentaires de MPI et utilisables sur *tout type* de machine, sont usitées [9]. Cela tient à notre volonté de portabilité : l'utilisation d'un sous-ensemble minimal de fonctions de MPI garantit que le noyau K/II fonctionne sur des machines où MPI n'était par disponible par défaut. En effet, l'effort pour développer une interface MPI suffisante pour supporter le noyau K/II est minimum et rapide [GT00]. Aucune ligne de code n'a à être modifiée pour qu'il puisse s'adapter à un nouvel environnement matériel et logiciel.

Un autre élément qui participe fortement à la portabilité du noyau est, comme nous l'avons déjà mentionné dans les sections précédentes, sa capacité à fonctionner sur des machines ne disposant pas de possibilités multi-tâches [10]. Bien que ces machines soient de moins en moins nombreuses, cette fonctionnalité lui permet de ne pas les écarter de son «champ d'action». Plus encore, il est envisageable de l'utiliser dans le cadre d'environnements dégradés (perte de processeurs pendant l'exécution) et surtout, l'autorise à fonctionner sur une cible non parallèle pour des besoins de validation algorithmique souple et rapide.

La représentation intermédiaire utilisée pour décrire l'application de manière compréhensible par le noyau a été de même conçue de manière à ne faire intervenir aucun élément architectural de la machine cible. Ainsi, elle reste la même quel que soit le support d'exécution, ce qui n'était pas complètement le cas dans les versions précédentes de SKiPPER qui prenaient en

munaut´e d'utilisateurs potentiels (en g´en´eral motiv´es par l'am´elioration des performances de leurs applications) qu'elle rebutait auparavant, mais elle n'en reste pas moins d´elicate en l'absence de m´ethodologie ad´equate car ces standards maintiennent l'utilisateur dans l'obligation de g´erer `a la fois ses probl`emes d'algorithmie et ceux inh´erents `a l'architecture cible et `a la programmation parall`ele en g´en´eral.

7. Ce point important lui permet d'ˆetre op´erationnel en quelques minutes sur de nombreuses machines sans aucun effort ni aucune connaissance pr´ealable de la machine de la part de l'utilisateur.

8. Pas forc´ement parall`ele d'ailleurs ! (ce qui permet un d´eboggage du code et une ´evaluation rapide et ais´ee de son fonctionnement dans l'environnement de d´eveloppement quotidien du programmeur.)

9. Voir l'annexe C pour la liste compl`ete des ressources MPI n´ecessaires au support du noyau K/II.

10. En l'occurrence, les capacit´es multi-tˆaches de la machine cible ne sont pas prises en consid´eration explicitement par le K/II, mais par l'interm´ediaire de la distribution MPI qui sera `a mˆeme, ou non, de d´emarrer plusieurs copies du noyau sur un mˆeme processeur. C'est le cas sur des machines `equip´ees d'un syst`eme d'exploitation multi-tˆaches comme UNIX. Mais ce peut ˆetre aussi le cas sur des machines n'ayant pas ce type de syst`eme, ou pas de syst`eme du tout (comme TransAlpha), mais o`u le processeur est directement `a mˆeme d'offrir cette possibilit´e (cas du T9000 par exemple).

compte à ce niveau l'architecture du réseau de processeurs. Cela est rendu possible ici par l'utilisation de l'interface MPI à laquelle est délégué le soin de la mise en œuvre des algorithmes de communication les plus appropriés entre deux processeurs pour une plate-forme donnée [11].

Le nouveau noyau de SKiPPER-II a donc été développé de manière à être indépendant de la machine cible, et de ce fait ne nécessite aucune modification avec le changement de cette dernière, là où la version précédente nécessitait encore la ré-écriture d'une partie du code. Mentionnons à ce propos que SKiPPER-II a été implanté avec succès sur les plate-formes matérielles suivantes (à des fins de développement d'applications parallèles et d'évaluation de ses propres performances) :

- machine parallèle spécifique (dédiée au traitement d'images) :
  - machine TransAlpha
    (processeurs DEC Alpha AXP (8) et Inmos Transputer T9000 (16), réseau de communication DS-Link)
- machines Beowulf :
  - machine de l'université Heriot-Watt d'Edimbourg en Ecosse
    (processeurs Intel Celeron (32), réseau de communication commuté Fast Ethernet)
  - machine Ossian de l'université Blaise-Pascal de Clermont-Ferrand
    (processeurs IBM/Motorola PowerPC G4 (4), réseau de communication Fast Ethernet)
- réseaux de stations de travail :
  - réseau de stations SUN Sparc IPC/IPX/LX (5) en Ethernet à 10 Mbits/s
  - réseau de stations SUN Ultra 5 (12) en Fast Ethernet
  - réseau de stations Silicon Graphics Indy/O2/Origin (8) en Fast Ethernet

La portabilité du noyau d'un environnement comme SKiPPER, condition nécessaire à la portabilité d'une application développée avec lui, participe à la facilité de déploiement *in fine* de cette application puisque qu'il permet au programmeur de rester dans le même environnement de programmation et de ne pas en changer sur chaque nouvelle machine. C'est un aspect essentiel de l'ergonomie d'un environnement d'aide à la programmation parallèle.

4.5 Extensibilité de la base de squelettes

L'introduction du «méta-squelette» TF/II dans SKiPPER-II offre une possibilité d'extension souple de la base de squelettes initiale, voire de son adaptation complète à d'autres problématiques que la vision.

En effet, le noyau de SKiPPER-II ne connaissant que le «méta-squelette» TF/II, il est *indépendant* de la base de squelettes utilisée : *l'introduction d'un nouveau squelette dans cette base*, ou la modification d'un ancien squelette, *ne demande aucune modification du noyau* [12].

11. L'expertise de la gestion du r´eseau est ainsi laiss´ee au concepteurs de la machine et aux d´eveloppeurs de l'interface MPI, et n'est donc plus int´egr´ee `a SKiPPER ce qui le rend ind´ependant des cibles et de leur ´evolution.

12. Sous r´eserve que le(s) squelette(s) puisse(nt) s'exprimer sous la forme d'un TF/II.

La couche exécutive de SKiPPER-II (le noyau) peut ainsi rester inchangée lors de l'ajout de nouveaux squelettes : c'est le frontal de SKiPPER qui se charge de faire la mise en correspondance entre un squelette et une représentation sous forme de TF/II. L'extension de la base peut ainsi être faite en fournissant simplement un schéma de mise en correspondance entre le nouveau squelette et le TF/II, sans qu'il n'y ait, de la part du concepteur d'un nouveau squelette, aucun code à fournir. Dans la précédente version, le concepteur d'un nouveau squelette aurait dû fournir le code exécutable de ce dernier, en plus de sa sémantique déclarative [13]. Avec SKiPPER-II, l'extensibilité de la base de squelettes ne concerne plus la partie exécutive, seulement le frontal. La pertinence pour le programmeur de l'ajout de certains schémas de parallélisation sous forme de squelettes peut donc être relativement facilement étudiée sous SKiPPER-II, le noyau n'ayant jamais a être retouché.

Non seulement ce mécanisme permet une extensibilité potentielle de la base de squelettes plus facile que dans la version précédente, mais de surcroît offre la possibilité de créer d'autres bases de squelettes plus adaptées à d'autres domaines applicatifs plus facilement. De plus, avec le même noyau, le programmeur d'applications peut utiliser indifféremment l'une ou l'autre de ces bases sans qu'il lui soit nécessaire ni de modifier le noyau, ni même de le recompiler : seule la partie applicative est à recompiler après un changement de base. De ce fait, SKiPPER-II peut potentiellement s'ouvrir sur d'autres domaines applicatifs que celui initialement prévu par le projet SKiPPER.

Avec cette technique, les possibilités d'extensibilité de la base de squelettes se rapprochent des approches de type *Design Patterns* discutés à la section 1.3.2.2 du chapitre 1 page 33.

4.6 Problèmes soulevés

4.6.1 Prédictabilité de performance

La prédiction des performances temporelles d'une application à partir de la simple durée d'exécution de ses fonctions séquentielles reste difficile avec SKiPPER-II étant donnée la nature purement dynamique de la distribution et de l'ordonnancement des processus et des communications. En effet les processeurs ne sont pas définitivement affectés à telle ou telle tâche par le noyau et peuvent jouer n'importe quel rôle (maître ou esclave) dans l'exécution d'un squelette. Notamment pour une même durée d'exécution des fonctions séquentielles de l'application, et pour un nombre de processeurs fixé, le temps d'exécution global dépendra de la disponibilité des ressources au moment où elles deviendront nécessaires. En effet le déploiement plus ou moins complet d'un squelette (fonction des ressources disponibles à cet instant), le temps d'interrogation du PLServer pour déterminer les ressources libres (proportionnel au nombre de processus à déployer), le niveau d'imbrication des squelettes (temps de transfert des données plus important), *etc*, participent à la difficulté de la prédiction. Plus le nombre de squelettes est grand et le niveau d'imbrication élevé pour un nombre de processeurs limité, plus les mécanismes cités plus haut vont influencer les performances globales et donc s'écarter des mesures données par les temps d'exécution des fonctions séquentielles. C'est une des conséquences de la souplesse d'utilisation de SKiPPER-II et de son adaptabilité à la cible.

13. Pour ˆetre tout-`a-fait pr´ecis, la donn´ee du code ex´ecutable n'´etait n´ecessaire avec la pr´ec´edente version que pour les squelettes dynamiques, le squelette statique SCM ´etant quant `a lui totalement d´efi ni en Caml puisque sa repr´esentation interm´ediaire sous la forme d'un GFDC d´ecoulait enti`erement de cette d´efi nition.

4.6.2 Gestion centralisée des ressources

Dans la version actuelle de SKiPPER-II, la gestion des ressources de calcul utilisées ou non est faite de manière centralisée dans le sens où les processus maîtres devant déployer leurs esclaves ne tiennent pas à jour individuellement une base de données locale qui les informerait en temps réel de la disponibilité de ces ressources. Ils font appel pour cela à un processus spécialisé qui gère une base de donnée centralisée et unique de la disponibilité desdites ressources. De ce fait les processus maîtres sont contraints d'interroger régulièrement le serveur de la base (PLServer) pour demander une ressource ou en libérer une. Si cette gestion est facile à mettre en œuvre, elle a été clairement identifiée des le départ comme un goulot d'étranglement potentiel. En effet, comme nous l'avons vu, le serveur monopolise une ressource de calcul et engendre des requêtes de petites tailles – mais nombreuses – sur le réseau de communication.

La mise en place d'une gestion distribuée de cette base n'est pas chose facile [CR95] [Dan99]. C'est un problème qui fait appel à des mécanismes complexes de maintien de la cohérence de la base pour être résolu [SE94] [KHP+95] [NPP98] [NP00]. A cela s'ajoute le fait que les processus demandeurs (les processus maîtres) dans ce cas de figure ne peuvent être associés véritablement à une copie de la base. En effet c'est une population de processus qui évolue en nombre et en localisation au cours du temps, et de manière non pré-déterminée à la compilation. La solution serait donc d'associer une copie de la base non pas à chaque processus maître, mais à chaque ressource (voire physiquement à un processeur). Le problème étant alors de pouvoir la mettre à jour régulièrement et de manière transparente pour les processus demandeurs et cela même sur un processeur ne supportant pas le multi-tâche. Une approche pourrait être d'implanter tout le mécanisme sous forme de gestionnaires d'interruptions sur chaque processeurs ; déclenchés sur modification de l'espace mémoire sous leur contrôle, ou régulièrement sur un signal d'horloge, ils pourraient tenir informés les autres processeurs des modifications de la base survenus localement. Cependant une telle approche serait fortement dépendante du matériel. C'est pourquoi une approche par requêtes semble plus propice à maintenir la portabilité de SKiPPER-II : toute requête locale en acquisition de ressources pourrait intégrer un mécanisme de mise à jour des bases distantes, uniquement dans le cas où la requête est satisfaite (pour ne pas surcharger le réseau).

4.6.3 Gestion du parallélisme dans les cas à faible nombre de ressources

Comme nous l'avons vu, un atout majeur de SKiPPER-II est de pouvoir gérer indifféremment les cas où le nombre de ressources pour l'exécution de l'application est suffisant et ceux dans lesquels il ne l'est pas, notamment celui où une unique ressource est disponible (émulation séquentielle).

Malheureusement cette souplesse se paie lorsque le nombre de processeurs physiquement disponibles est faible (de l'ordre de 2 à 4) et qu'ils ne disposent pas de possibilités de multi-tâche. En effet, dans ce cas, certains processus maîtres vont monopoliser des unités de traitement pour leur seul fonctionnement sans que celles-ci puissent être affectées à un traitement réel (fonction utilisateur de calcul). La figure 4.14 montre ainsi ce qui se passe sur un petit exemple. L'arbre qui y est représenté est le schéma de l'application. Deux squelettes TF/II y sont imbriqués (les deux instances du squelette imbriqué ont été distinguées par des numeros différents pour plus de lisibilité). L'architecture proposée est constituée de seulement 4 processeurs. Sur chacun d'eux est installée une unique copie du noyau. Le schéma montre, sous chaque processeur symbolisé, la liste des processus affectés à chaque copie du noyau. On a supposé pour les besoins de l'exemple que l'exécution de l'esclave S2A est presque aussi longue que l'exécution des esclaves S3A et S3B réunis (cas le plus défavorable). S2B est mis entre parenthèses pour indiquer que son exécution n'a pas lieu en même temps que les trois autres esclaves : S2A, S3A et S3B ont une exécution quasi-simultanée, alors que S2B est pratiquement seul a s'exécuter au moment de son activation. La figure 4.15 donne le diagramme temporel de l'activation de ces processus. Ces figures montrent clairement que les deux processeurs auquels sont affectés les processus maîtres (*Mx*) passent leur temps à attendre la fin d'exécution de leurs esclaves, et donc ne participent pas véritablement au travail proprement dit ; alors que dans le même temps l'exécution des esclaves (*Sxy*) en vient à être, en partie, séquentialisée.

Dans le cas de processeurs disposant de capacités de multi-tâche, le problème persiste mais ne se pose pas dans les mêmes termes. Ici l'exécution d'un maître peut être «recouverte» par l'exécution d'un esclave, mais dans ce cas le point délicat est le choix du nombre d'esclaves qu'on souhaite pouvoir affecter à chaque processeur et du nombre de ressources qui lui sont affectées. Normalement, un seul esclave devrait être autorisé à s'exécuter sur un même processeur, ce qui peut être décidé par le PLServer, mais il n'en reste pas moins que la distribution optimum des esclaves et des maîtres sur les processeurs disposants de plusieurs ressources est un problème difficile, non pris en compte actuellement par SKiPPER-II. De ce fait, le nombre de ressources allouées à chaque processeur est laissé à la discrétion de l'utilisateur qui peut la modifier directement au moment de l'exécution de son application (sans recompilation ou modification du code). Mais cette possibilité reste cependant un paramètre délicat à régler. La figure 4.16 illustre le cas du multi-tâche sur le même schéma d'application que pour l'exemple précédent. Ici aussi les deux instances du TF/II imbriqué sont numérotées différemment afin de les distinguer. L'utilisation dans cet exemple du même nombre de processeurs que précédemment, mais de deux copies du noyau par processeur, fait qu'il y a suffisamment de copies disponibles pour que tous les squelettes (3 au total) puissent se déployer complètement et simultanément. Ainsi, sur la figure, on peut voir que trois copies jouent le rôle des trois processus maîtres, et que quatre autres servent d'esclaves. Les esclaves sont ici affectés à des copies qui se trouvent sur des processeurs dont les autres copies ne supportent pas l'exécution d'autres esclaves, tout au plus celles de processus maîtres. La figure 4.17 donne quant à elle le diagramme d'activation des processus en question pour la configuration choisie.

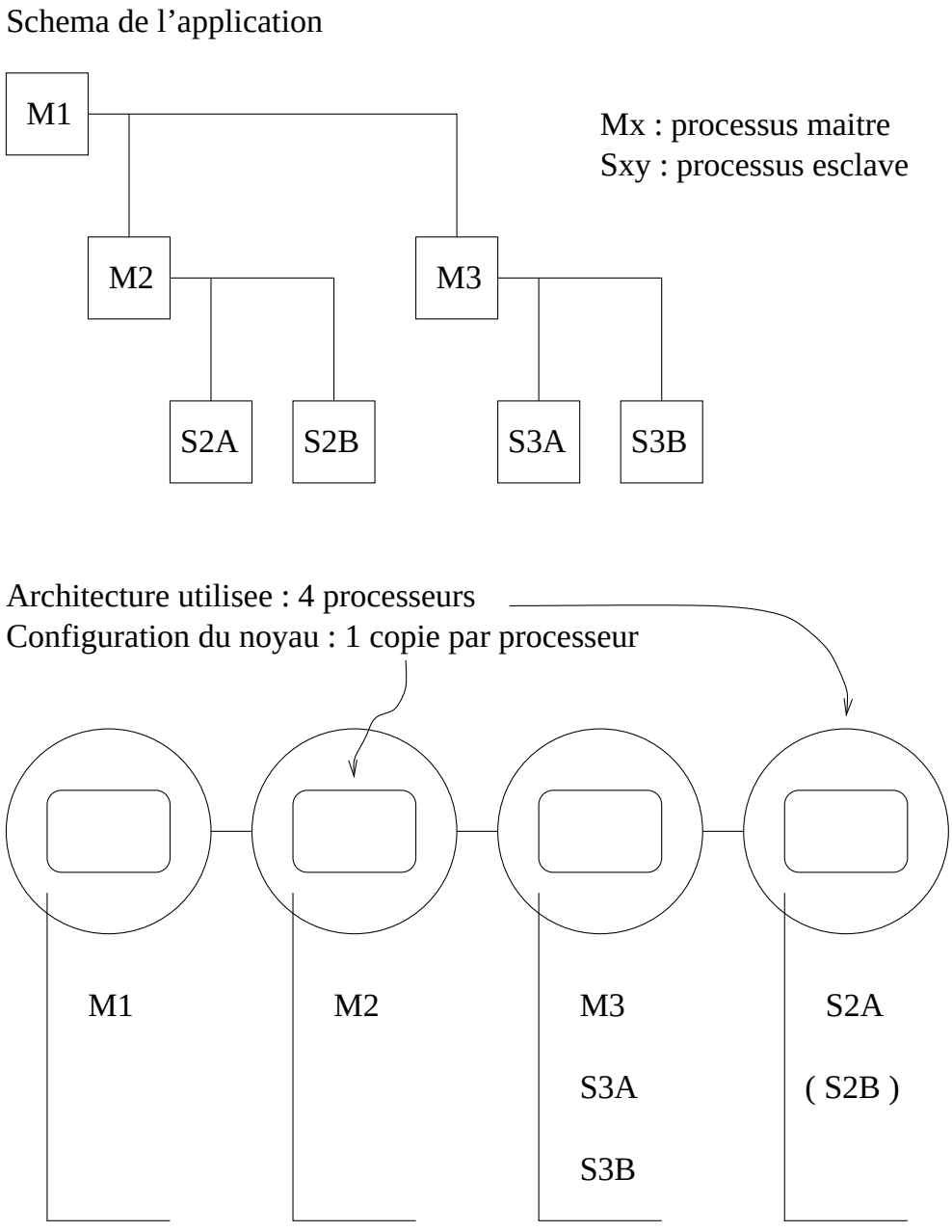


FIG. 4.14 – *Placement des processus pour un nombre de ressources très faible.*

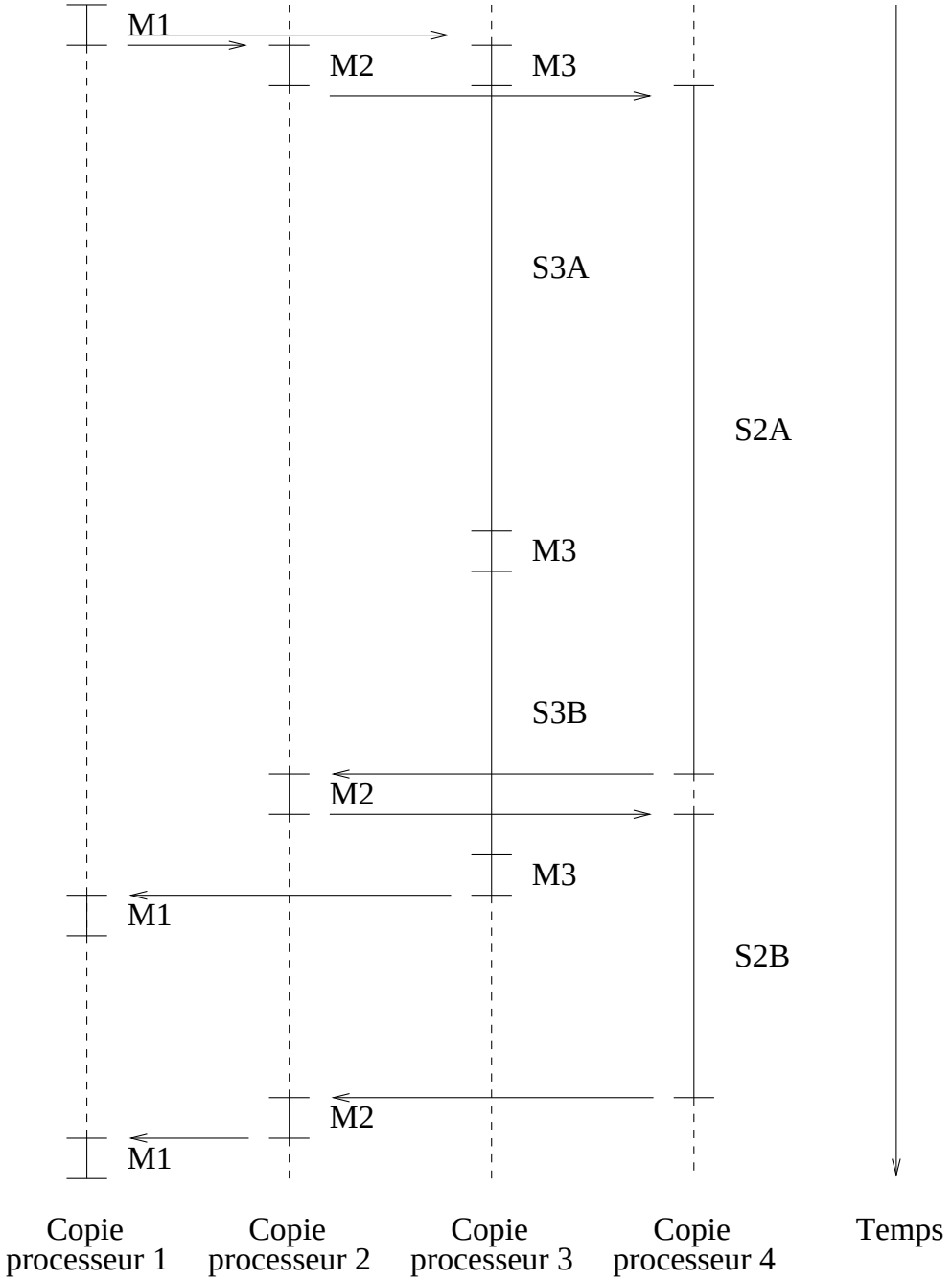


FIG. 4.15 – *Diagramme temporel d'activation des processus pour un nombre de ressources très faible.*

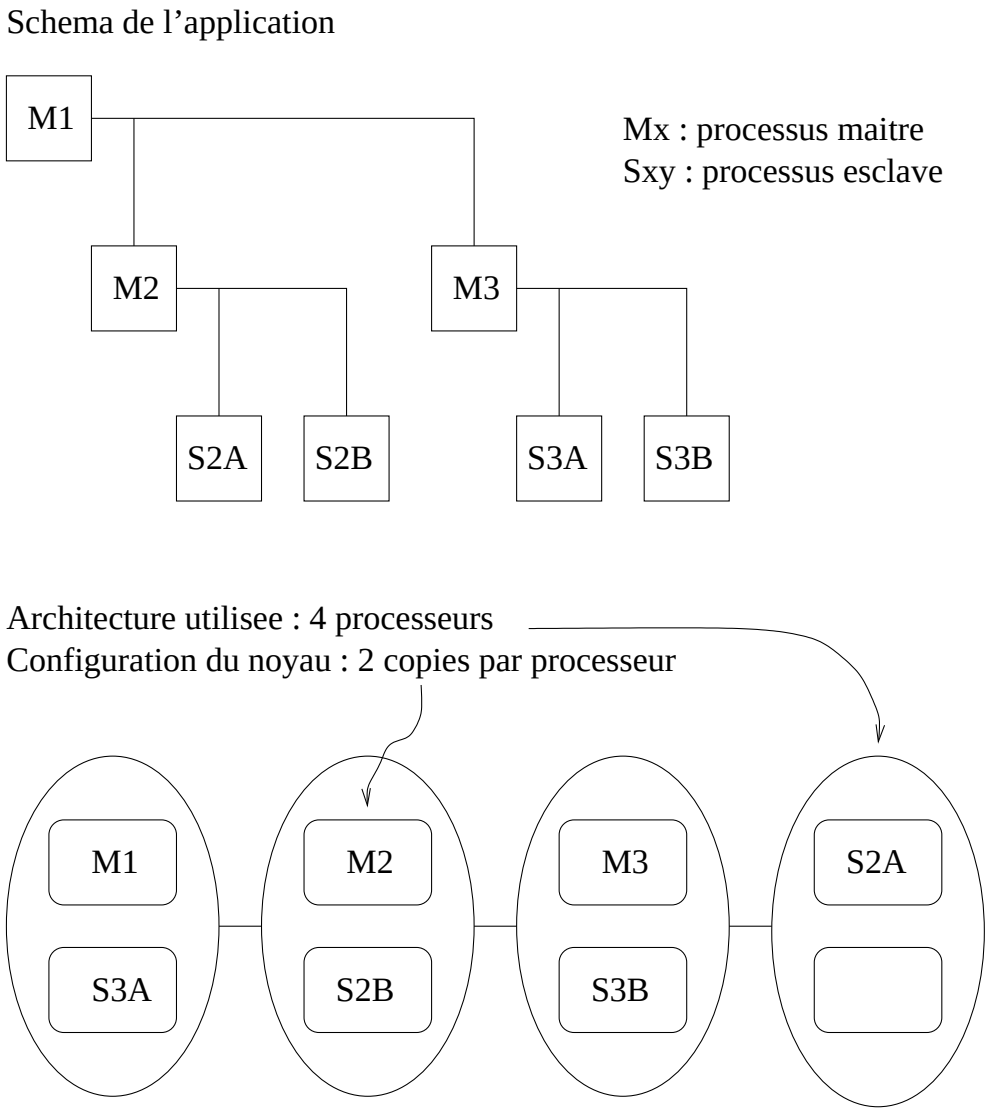


FIG. 4.16 – *Placement des processus pour un nombre de processeurs limités, mais avec possibilités de multi-tâche.*

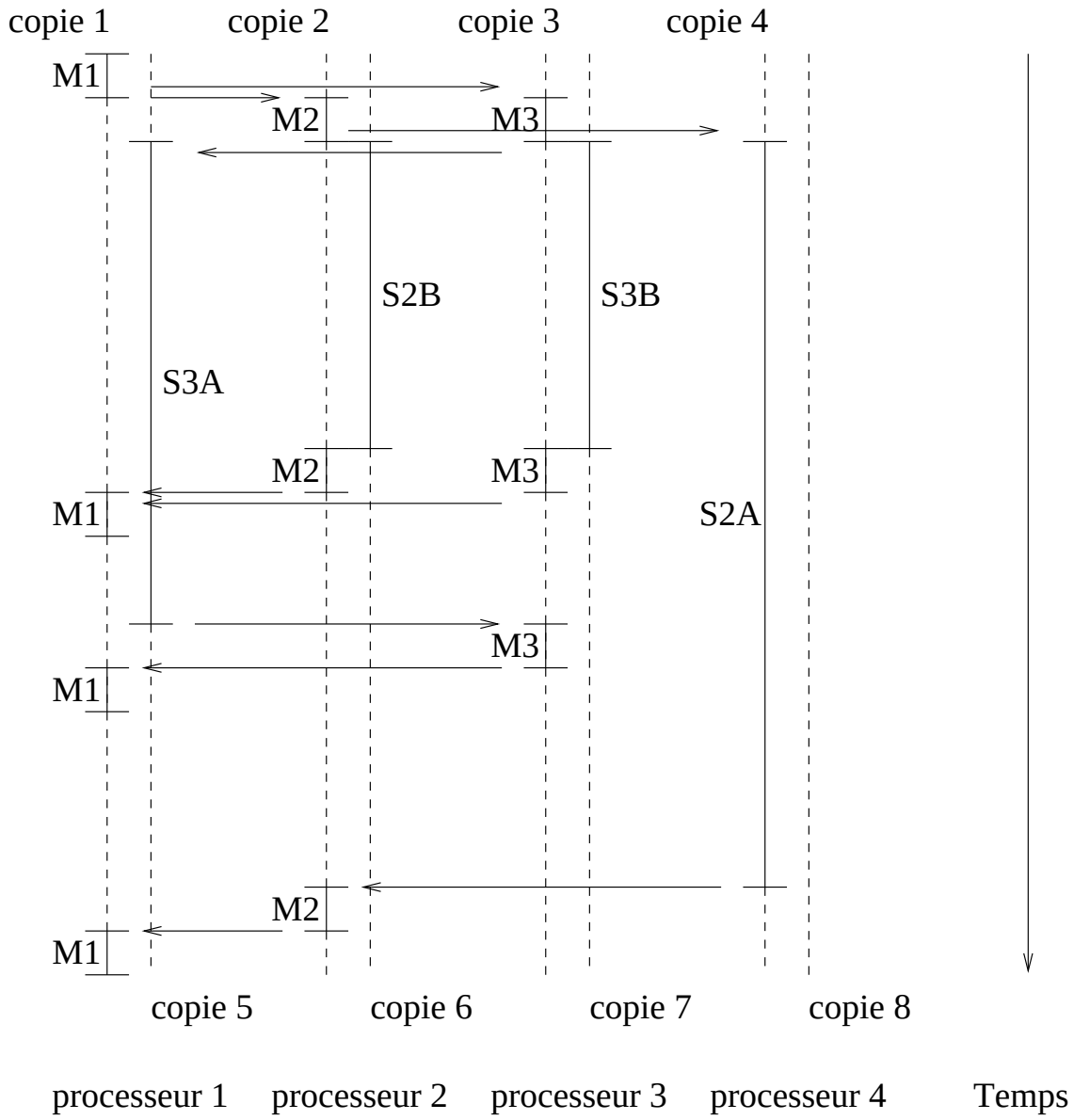


FIG. 4.17 – *Diagramme temporel d'activation des processus pour un nombre de processeurs limités, mais avec possibilités de multi-tâche.*

4.7 Conclusion

Le développement de SKiPPER-II découle de la volonté de proposer une version de l'environnement pouvant prendre en charge l'imbrication de squelettes algorithmiques. Pour ce faire, le modèle d'exécution essentiellement «statique» de la version précédente a été abandonné au profit d'un modèle complètement «dynamique». En effet, SKiPPER-I différenciait les squelettes «statiques» des squelettes «dynamiques» parce qu'il fonctionnait avec un modèle statique d'exécution, alors que certains des squelettes ont un comportement par nature dynamique. SKiPPER-II élimine cet aspect pour ne gérer qu'un seul et unique modèle d'exécution : le modèle dynamique. Nous avons choisi ce modèle car le modèle statique ne permettait pas à lui seul de rendre compte du comportement de tous nos squelettes, et empêchait de ce fait l'obtention d'une représentation homogène des squelettes. Or cette représentation est nécessaire pour autoriser une composition «régulière» des squelettes, c'est-à-dire se faisant toujours de la même façon quelles que soient la composition et les types de squelettes qui interviennent, en l'absence de toute exception dans la description d'une composition particulière.

Un «méta-squelette» (TF/II) a ainsi été proposé afin de facilité l'imbrication en rendant les aspects de composition plus homogènes. SKiPPER-II se fonde de ce fait sur un noyau chargé d'exécuter les squelettes des applications après leur mise sous la forme de TF/II.

Bien entendu le revers de cette approche est d'ajouter aux squelettes intrinséquement statiques (comme le SCM) un certain coût dû à leur gestion «à la volée» qui pourrait être supprimée puisque leur comportement au cours du temps (et spatialement) peut être complètement prédit. Mais cette surcharge peut être réduite en utilisant un schéma de communication adéquat comme cela est fait dans SKiPPER-II.

Les avantages principaux de cette approche sont donc :

- une gestion identique de tous les squelettes («statiques» et «dynamiques»),
- des possibilités de composition systématiques des squelettes,
- la gestion dynamique des processeurs disponibles [14],
- une émulation séquentielle directe (plus de distinction avec une exécution parallèle),
- une grande portabilité de l'environnement,
- une extensibilité de la base des squelettes facilité,
- la compatibilité avec la version précédente de SKiPPER.

Le chapitre suivant illustre et valide l'approche retenue pour SKiPPER-II à travers l'expérimentation d'applications de complexité variable. Le comportement de SKiPPER-II y est quantifié et comparé à celui de SKiPPER-I.

14. Cette caract´eristique offre notamment, par l'interm´ediaire du PLServer (voir la section 4.3.4.1 page 103), la possibilit´e d'envisager `a terme un certain niveau de «tol´erance aux fautes» pour SKiPPER-II.

Chapitre 5

Résultats expérimentaux

Ce chapitre est consacré à l'évaluation de l'environnement SKiPPER-II. Différents algorithmes utilisant l'imbrication de squelettes sont utilisés. SKiPPER-II est comparé avec la première version de l'environnement et quelques environnements de programmation parallèle similaires.

5.1 Aperçu des applications

Nous présentons dans ce chapitre un panel d'applications afin d'évaluer le comportement de l'environnement SKiPPER-II. Ces applications ont chacune une spécificité permettant d'apprécier des propriétés différentes de l'environnement de programmation parallèle. Elles peuvent être classées en cinq catégories :

- vérification du fonctionnement élémentaire et de la compatibilité ascendante :
  - calcul d'histogramme (section 5.2.2),
  - détection de taches lumineuses (section 5.2.3),
  - division récursive d'images (section 5.2.4) ;
- comparaison avec une implantation manuelle :
  - imbrication simple de deux squelettes (section 5.3) ;
- comportement en cas d'imbrication :
  - imbrication simple de deux squelettes (section 5.3),
  - multiplication matricielle d'entiers de longueur arbitraire (section 5.4) ;
- comparaison avec un autre environnement de programmation parallèle :
  - multiplication matricielle d'entiers de longueur arbitraire (section 5.4) ;
- étude d'un cas réel d'algorithme nécessitant une imbrication de squelettes :
  - suivi de visages (section 5.5) ;

5.2 Fonctionnement élémentaire et comparaison avec SKiPPER-I

5.2.1 Préambule

Nous présentons dans les trois sous-sections qui suivent des programmes dont l'objet est d'évaluer le comportement de SKiPPER-II pour la mise en œuvre de chacun de ses squelettes pris individuellement. Cette démarche a aussi pour objectif de donner quelques éléments de comparaison avec la version SKiPPER-I de l'environnement dans la mesure où les algorithmes sont les mêmes que ceux qui avaient été choisis pour tester le fonctionnement de la première version [Gin99].

Concernant ce dernier aspect, nous avons mené deux séries de tests.

La première s'est déroulée en implantant les algorithmes proposés pour l'évaluation de SKiPPER-I directement avec SKiPPER-II. Nous présentons les résultats obtenus seulement pour le squelette SCM. En effet, ne prenant pas en compte la différence de puissance entre les processeurs actuels et les processeurs utilisés en 1998 pour les tests de SKiPPER-I, les résultats obtenus ne permettent pas une comparaison significative [1]. Nous présentons néanmoins le comportement obtenu pour le SCM car il permet de tirer quelques enseignements sur le comportement de SKiPPER-II dans le cas où le ratio calcul/communication est très faible, et donc défavorable à une parallélisation de l'algorithme. Les résultats fournis par la mise en œuvre des squelettes DF et TF ne présentant pas plus d'intérêts que ceux attachés au squelette SCM, nous avons préféré livrer directement les résultats obtenus après correction des algorithmes pour tenir compte des performances actuelles des processeurs.

La correction que nous avons apporté au programme de test pour tenir compte des différences de performance entre les processeurs consiste à augmenter le nombre de calculs effectués, sans modifier le volume des communications [2]. Pour ce faire, les opérations contenues dans la fonction utilisateur de traitement de chaque squelette sont répétées plusieurs fois au sein de chaque esclave. Le nombre de répétitions a été fixé pour obtenir sensiblement les mêmes performances sur un seul processeur que celles obtenues pour le test de SKiPPER-I [Gin99].

L'évaluation de SKiPPER-II avec ces programmes s'est faite sur une machine Beowulf équipée de 32 nœuds de calcul de type Intel Celeron cadencés à 533 MHz. Le Beowulf était équipé d'un réseau commuté Fast Ethernet à 100 Mbits/s.

1. Les processeurs utilis´es dans la machine ayant servie `a l'´evaluation de SKiPPER-I ´etaient des Inmos Transputer T9000E, cadenc´es `a 20 MHz, d´eveloppant 100 Mips cr^ete et 10 Mflops cr^ete. Ceux utilis´es dans le Beowulf nous ayant servi pour les tests de SKiPPER-II sont des Intel Celeron, cadenc´es `a 533 MHz, d´eveloppant 1445 Mips cr^ete et 717 Mflops cr^ete.

2. Les r´eseaux de communication utilis´es dans les deux machines de test offrant des bandes passantes similaires (DS-Link 100 Mbits/s pour le r´eseau d'interconnexion des T9000E, et Ethernet 100 Mbits/s pour le r´eseau d'interconnexion des Celeron), c'est essentiellement la diff´erence relative de vitesse de traitement par rapport au d´ebit des r´eseaux qui est signifi cative.

5.2.2 Un algorithme pour le SCM : l'histogramme. Comparaison avec SKiPPER-I

Cette section présente la mise en œuvre simple du squelette à parallélisme de données SCM, sans imbrication. Le programme retenu est un calcul d'histogramme des niveaux de gris d'une image. Du fait de la régularité des traitements, cet algorithme se prête bien à son exécution sous la forme d'un SCM.

La description fonctionnelle en Caml de cette application est la suivante (description utilisée par le frontal de SKiPPER) :

```
let image       = lecture_image 512 512
let histogramme = scm division calcul fusion image

Signature des fonctions (Caml) :
division :    image
           -> image tuple
calcul   :    image
           -> histo
fusion   :    (histo tuple)
           -> histo

Prototypes des fonctions (C) :
void division( image * , image ** );
void   calcul( image * , histo *  );
void   fusion( histo **, int   *  );
```

Dans cet exemple, c'est une image de 512 pixels par 512 pixels qui est traitée. La fonction `lecture_image` a pour rôle de fournir l'image à traiter à partir d'une entrée par caméra vidéo par exemple. La fonction `division` décompose l'image en plusieurs bandes horizontales (autant que de processeurs disponibles). Le traitement proprement dit (calcul de l'histogramme) est fait, sur chaque bande, par la fonction `calcul`. Enfin la fonction `fusion` combine les histogrammes partiels issus de la fonction de calcul pour fournir le résultat sur l'image entière.

Nous présentons ci-après les résultats de l'expérimentation en reproduisant directement ce qui avait été fait dans le cadre de SKiPPER-I (figures 5.1 et 5.2). Dans un second temps sont présentés les résultats pour la même expérimentation, mais en augmentant les volumes de calcul pour compenser la différence de puissance des processeurs utilisés actuellement par rapport à ceux de 1998. Dans ce dernier cas nous les comparons à ceux obtenus pour SKiPPER-I.

Comme nous l'avons mentionné précédemment, les mauvais résultats en terme d'accélération obtenus par une implantation directe (sans tenir compte de la puissance des processeurs) de l'algorithme utilisé pour le test de SKiPPER-I est dû à la trop faible quantité d'opérations qui sont faites. Ici le réseau de communication s'impose. Au delà de 8 processeurs les temps d'exécution augmentent. A partir de ce nombre de processeurs les communications et la gestion des esclaves deviennent prépondérantes : avec un seul processeur, les calculs représentent près de 90 % du temps total d'exécution (très peu de communication et surtout de gestion des esclaves), entre 2 et 8 processeurs le taux chûte entre 8 % et 13 % seulement. L'augmentation

du nombre de processeurs ne permet plus un gain de temps au niveau de la parallélisation des calcul, mais accroît en revanche considérablement les communications de gestion des esclaves puisque leur nombre a augmenté. Il est aussi intéressant de constater que, dans cette situation, même si la courbe de comportement de l'environnment est similaire à celles observés dans d'autres de nos tests (voir sections suivantes), mais contrairements à ces derniers, elle ne passe jamais en dessous du temps d'exécution sur un seul processeur (en émulation séquentielle). Il est clair qu'avec les performances des processeurs utilisés, il n'est pas judicieux de paralléliser cet algorithme en l'état. Une exécution séquentielle est plus rapide dans la mesure où, en parallèle, les processeurs vont passer le plus clair de leur temps à attendre les données plutôt qu'à les traiter.

C'est pourquoi nous présentons maintenant le même test mais corrigé pour tenir compte de la puissance des processeurs.

Le comportement obtenu après correction est le comportement type du noyau de SKiPPER-II pour l'exécution d'un squelette SCM (cf. figures 5.3 et 5.4). En effet, on constate dans un premier temps une augmentation du temps d'exécution pour deux processeurs par rapport au temps d'exécution en émulation séquentielle sur un seul et unique processeur. Cette hausse est due à une augmentation des communications inter-processeurs, sans gain réel d'efficacité au niveau de l'aspect calculatoire dans la mesure où le processeur supplémentaire mis en œuvre dans cette configuration n'est pas exploité pour le traitement des données, mais seulement comme répartiteur des tâches (processus maître). Ensuite le temps d'exécution décroît régulièrement avec l'augmentation du nombre de processeurs. On note une remontée du temps d'exécution pour un grand nombre de processeurs [3] (ici supérieur à 15). Cette perte d'efficacité traduit la mise en œuvre d'un nombre trop important de processeurs : ils n'ont plus une charge de calcul suffisante avec un découpage de l'image trop poussé. Il en résulte donc une prédominance des communications et de la gestion de esclaves face au temps de calcul effectif sur chaque unité de traitement qui passe la majorité de son temps à attendre et à ré-émettre ses données.

On peut aussi noter que le temps d'exécution atteint par cette version du programme de test avec l'augmentation du nombre de processeurs (35 ms pour 20 processeurs) est le même que celui que le programme obtient dans sa version non corrigée [4]. Cela confirme que les opérations réalisées à ce stade (20 processeurs et plus) sont principalement des opérations de gestion des esclaves et non de traitement des données.

Comparée à la courbe obtenue pour SKiPPER-I, la courbe d'exécution corrigée présente une même dynamique en termes de temps d'exécution, mais avec une décroissance du temps moins rapide pour SKiPPER-II, ce qui aboutit à une courbe d'accélération moins bonne.

3. Cet effet n'avait pas ´et´e observ´e lors des tests de SKiPPER-I pour lesquels le nombre de processeurs ´etait toujours inf´erieur `a 8.

4. Les deux versions pr´esentent `a ce niveau une augmentation lente mais r´eguli`ere de leur temps d'ex´ecution, mais celle-ci ´evolue dans les mˆemes proportions avec le nombre de processeurs dans les deux cas (la brusque ascension de la courbe de la fi gure 5.1 n'est due qu'`a l'´echelle utilis´ee pour sa repr´esentation).

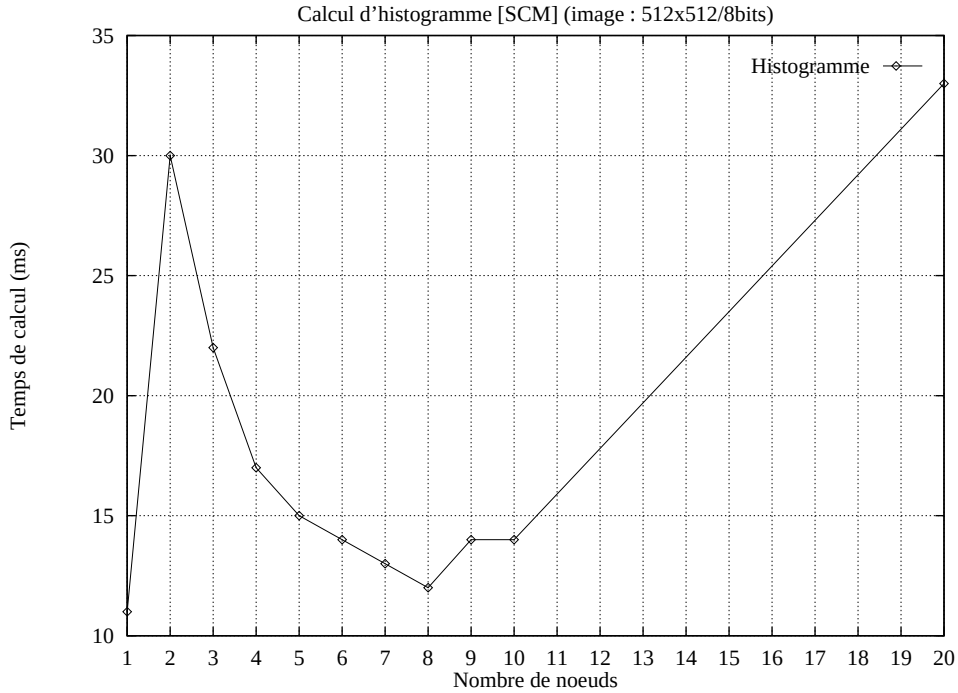


FIG. 5.1 – *Temps d'exécution pour l'histogramme (sans correction).*

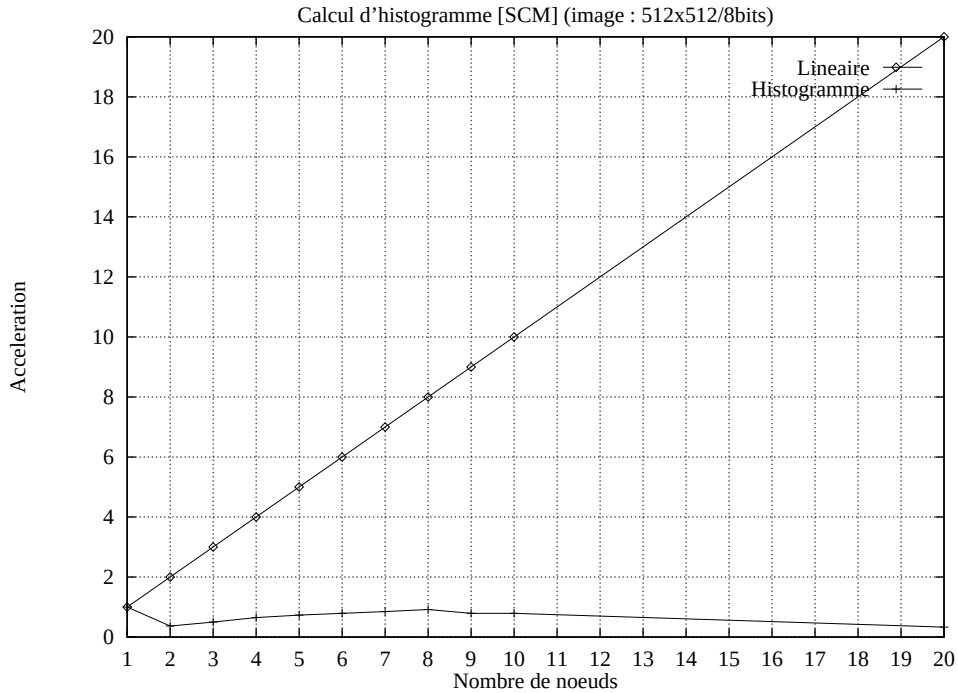


FIG. 5.2 – *Accélération pour l'histogramme (sans correction).*

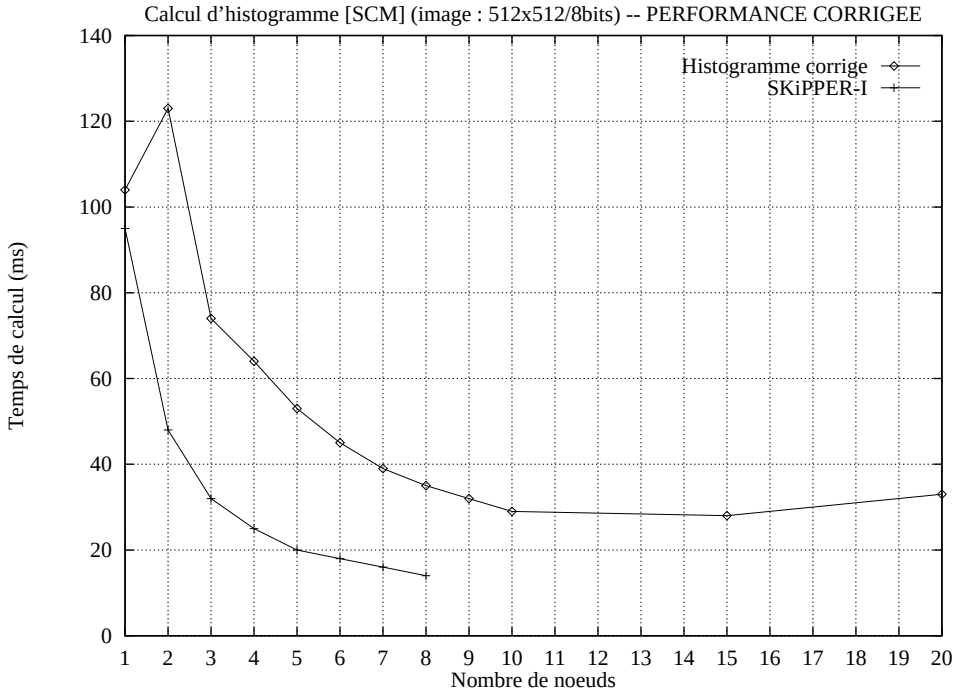


FIG. 5.3 – *Temps d'exécution pour l'histogramme (avec correction).*

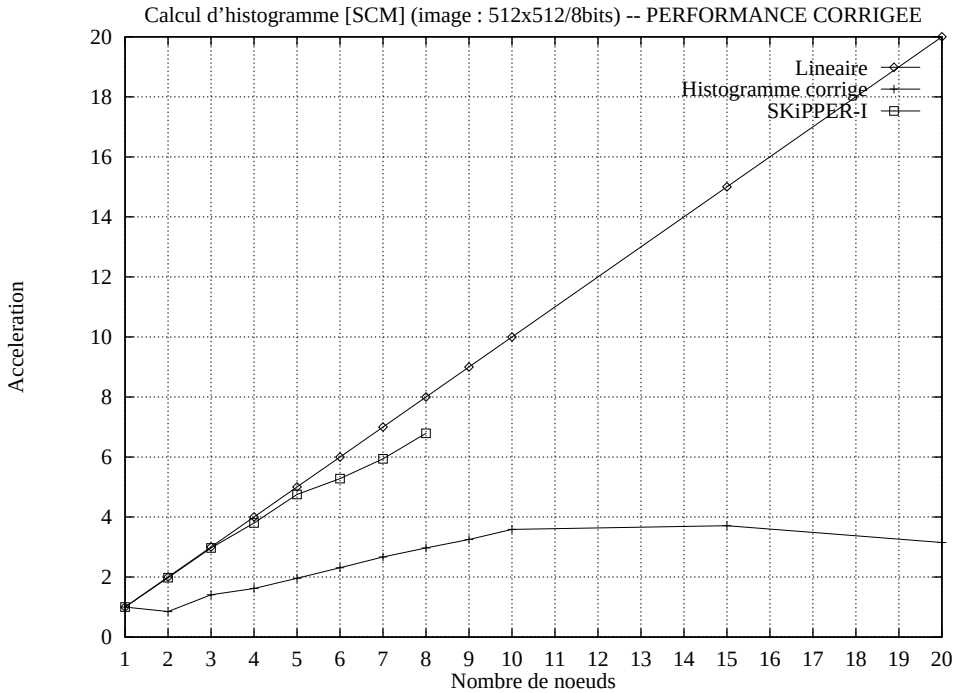


FIG. 5.4 – *Accélération pour l'histogramme (avec correction).*

5.2.3 Un algorithme pour le DF : détection de taches lumineuses dans une image. Comparaison avec SKiPPER-I

Cette section présente la mise en œuvre du squelette DF en dehors de toute imbrication. Le programme retenu consiste à détecter dans une image des taches lumineuses. L'intérêt de ce type de détection est par exemple la localisation d'amers visuels placés sur des véhicules afin de les identifier et de les suivre dans un fbt d'images vidéo [MCMD98] [Mar00].

En entrée le squelette DF prend une liste de fenêtres d'intérêt de tailles variables. Chacune d'elles contient un certain nombre de taches lumineuses (ce nombre pouvant éventuellement être nul). Dans chaque fenêtre l'algorithme de détection réalise les opérations suivantes. Il commence par désigner des pixels pouvant potentiellement faire partie d'une tache. Pour ce faire une partition des pixels est opérée avec comme critère une valeur de luminosité seuil au dessus de laquelle le pixel est considéré comme point candidat pour être un élément constitutif d'une tache. A partir de là, ces points sont agrégés aux taches déjà existantes. Si cela ne peut être le cas (il fait partie d'une tache non encore détectée), alors une nouvelle tache est générée dans la liste des taches déjà détectées. Les taches sont modélisées sous la forme d'une structure de données comprenant:

- le nombre de pixels constituant la tache,
- les coordonnées de son centre de gravité,
- les coordonnées du rectangle englobant.

De ce fait, un point candidat sera agrégé à une tache s'il est connexe au rectangle qui englobe cette tâche. Le résultat final est constitué d'une liste de taches lumineuses (voir la figure 5.5).

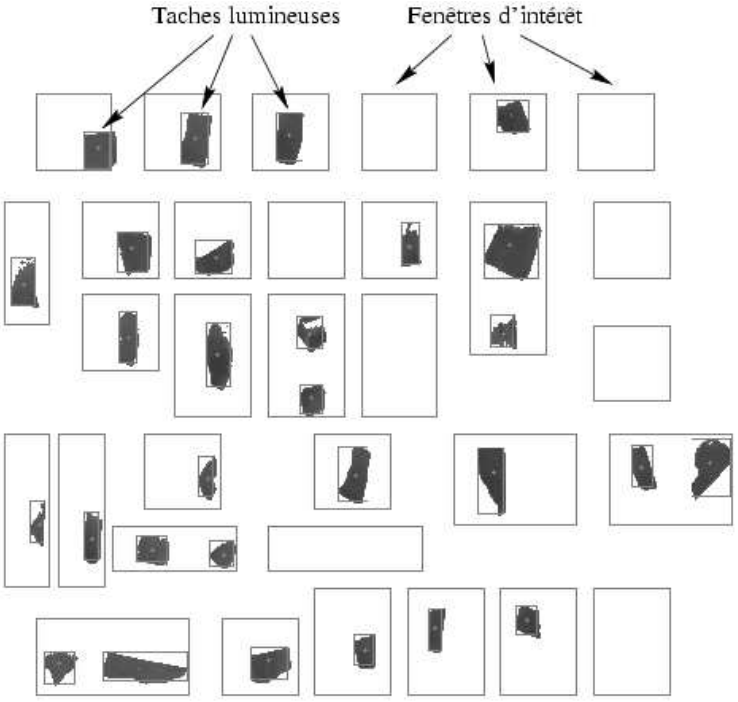


FIG. 5.5 – *Exemple de détection de taches lumineuses dans une image.*

Dans le cadre de cette expérimentation, les fenêtres d'intérêt ne changent ni en taille ni en position (contrairement à ce qui se passe dans un algorithme complet de détection d'amers visuels comme [MCMD98]). L'intérêt d'un schéma de parallélisation comme celui du DF pour ce type d'algorithme est que le temps de calcul nécessaire au traitement d'une fenêtre d'intérêt est dépendant de la taille de ces fenêtres et du nombre de taches qui y sont incluses. D'où la nécessité d'utiliser un squelette avec distribution dynamique des données, et donc le squelette DF.

La spécification fonctionnelle en Caml pour ce programme est la suivante :

```
let image    = lecture_image 512 512
let fenetres = extraction_fenetres image
let taches   = df detection_taches accumuler_taches
                  liste_initiale fenetres

Signature des fonctions (Caml) :
extraction_fenetres :    image
                       -> image list
detection_taches    :    image
                       -> tache list
accumuler_taches    :    tache
                       -> tache list
                       -> tache list

Prototypes des fonctions (C) :
void extraction_fenetres( image *, image **           );
void    detection_taches( image *, tache **           );
void    accumuler_taches( tache  , tache * , tache ** );
```

Dans cet exemple une image de 512 pixels par 512 pixels est traitée sur 8 processeurs. La fonction `lecture_image` a pour rôle de fournir l'image à traiter à partir d'une entrée par caméra vidéo par exemple. La fonction `extraction_fenetres` permet de constituer une liste de fenêtres d'intérêt pour les traitements à partir de l'image. La fonction `detection_taches` détecte les taches présentes dans une fenêtre d'intérêt et retourne une liste (éventuellement vide) de taches lumineuses. Grâce à `accumuler_taches` le programme constitue une liste globale des taches détectées, et ceci pour toute l'image. Enfin `liste_initiale` fournit une liste initiale vide de taches.

Le même phénomène que celui constaté dans pour le test précédent (SCM) sans correction se reproduisant ici, pour les mêmes raisons, nous présentons aux figures 5.6 et 5.7 les résultats de l'expérimentation après correction de la différence de puissance entre les processeurs.

Dans cette version corrigée, la courbe de comportement en temps d'exécution se rapproche de celle obtenue avec un squelette SCM. La différence majeure se situe au niveau de la remontée avec le nombre de processeurs qui est plus brusque et importante.

On peut aussi noter que le nombre de processeurs «optimum», qui était de 3 dans l'implantation non corrigée (temps d'exécution le plus faible pour un nombre de processeurs supérieur à un), s'est décalé vers 6 dans cette version. Cela est dû à l'augmentation du volume de calcul qui doit être supportée par davantage de processeurs.

On remarquera enfin une diminution du temps d'exécution pour plus de 10 processeurs (présente aussi dans la version non corrigée), pour 16 fenêtres d'intérêt uniquement. Cette diminution n'a pas d'explication si ce n'est l'incertitude sur la mesure. Elle traduit ainsi non pas une baisse mais seulement une stabilisation du temps d'exécution, visible sur les courbes retraçant les temps pour moins de fenêtres d'intérêt.

Comparées aux courbes obtenues pour SKiPPER-I (reproduites aux figures 5.8 et 5.9), les courbes d'exécution corrigées présentent des temps d'exécution comparables (même dynamique à nombre de fenêtres donné). SKiPPER-II présente par contre un très mauvais comportement après 6 processeurs pour plus de 40 fenêtres. Nous ne pouvons cependant que difficilement comparer les deux solutions étant donné que SKiPPER-I n'a été testé que jusqu'à 8 processeurs. A cela s'ajoute le fait que l'augmentation du volume de calcul conduit à une inversion de l'évolution de la courbe de temps de calcul (accroissement de ce temps) pour un nombre de processeurs augmentant avec ce volume. Le mauvais comportement dénoté plus haut peut ainsi être gommé avec des volumes de calcul encore plus grands.

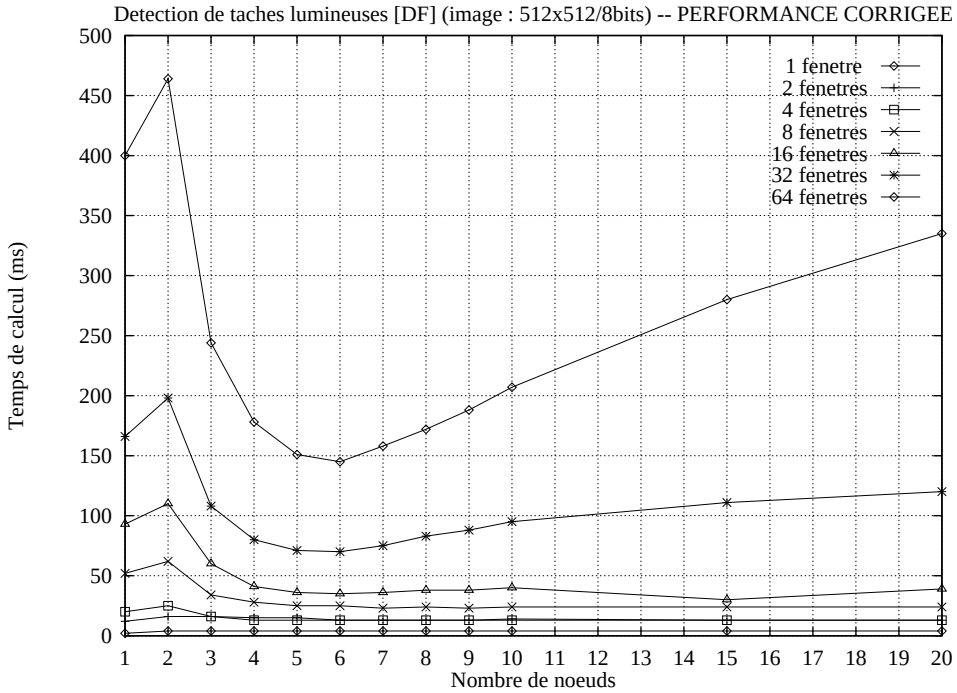


FIG. 5.6 – *Temps d'exécution pour la détection de taches lumineuses (avec correction).*

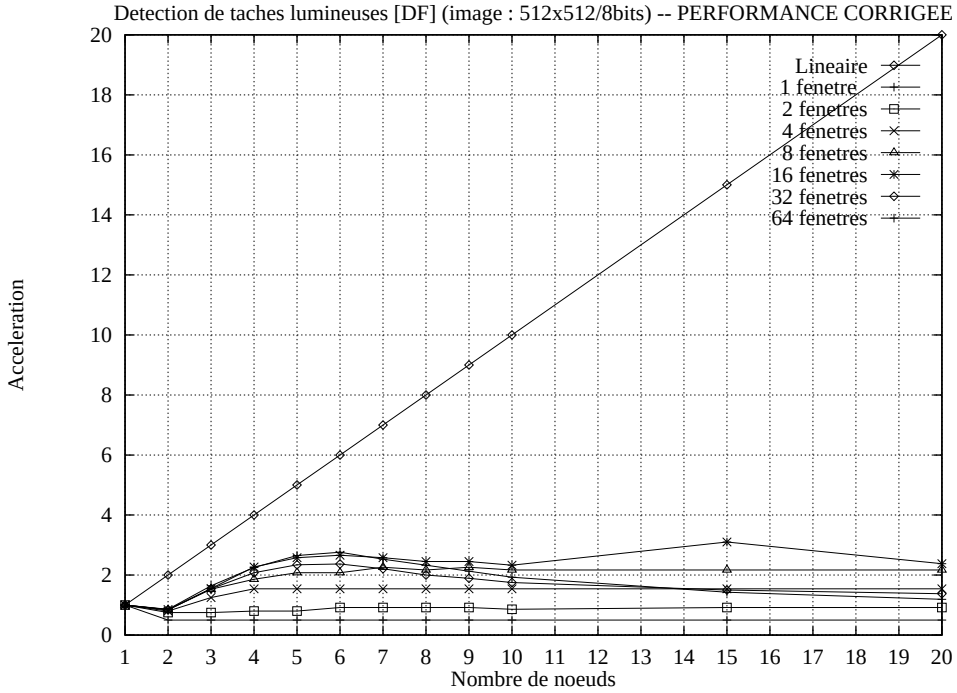


FIG. 5.7 – *Accélération pour la détection de taches lumineuses (avec correction).*

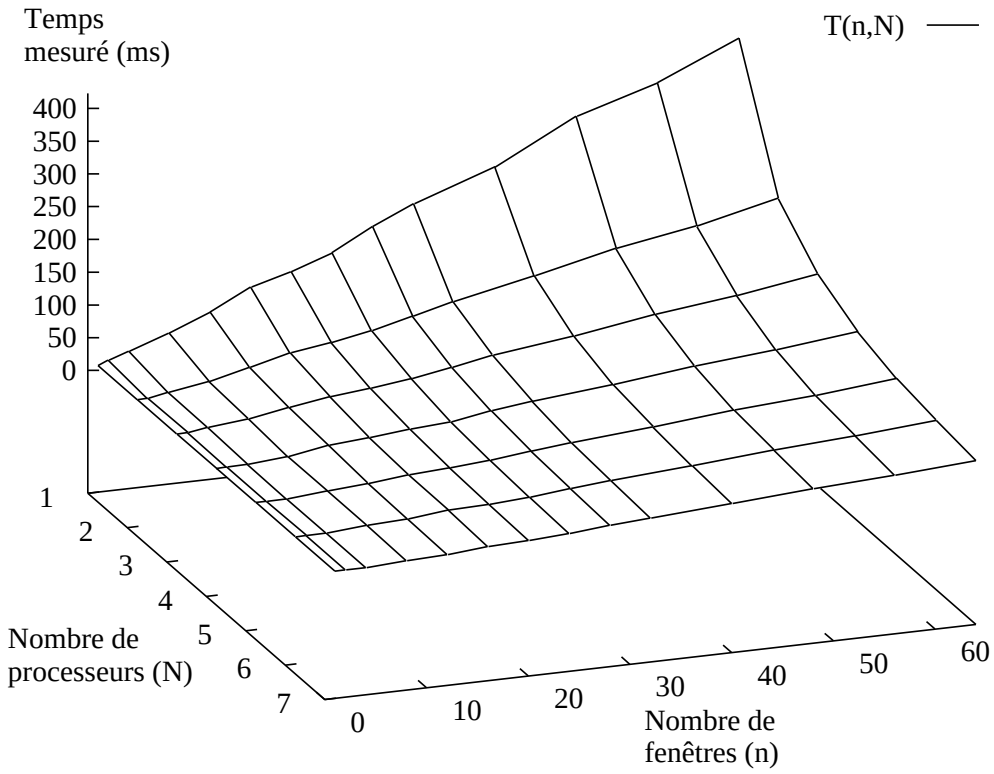


FIG. 5.8 – *Temps d'exécution pour la détection de taches lumineuses avec SKiPPER-I (extrait de [Gin99]).*

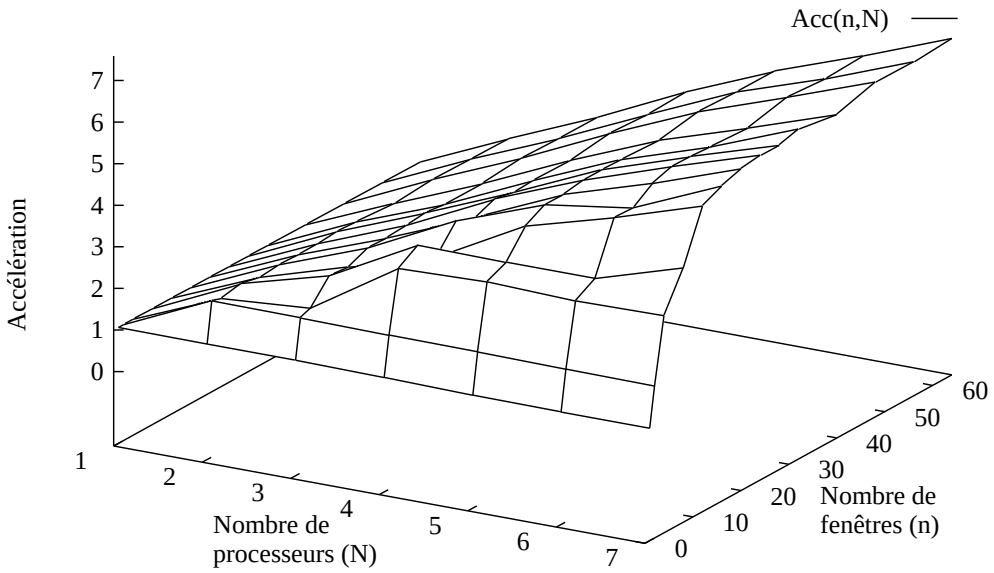


FIG. 5.9 – *Accélération pour la détection de taches lumineuses avec SKiPPER-I (extrait de [Gin99]).*

5.2.4 Un algorithme pour le TF : division récursive d'images. Comparaison avec SKiPPER-I

Est présenté dans cette section un algorithme de type division récursive permettant d'exploiter le squelette TF, ici sans imbrication. Ce genre d'algorithme est utilisé notamment dans les applications de segmentation d'images par «division-fusion». On met alors en œuvre un critère d'homogénéité sur les niveaux de gris, sur le mouvement ou bien encore sur la texture [HP74] (exemple de division récursive sur la figure 5.10).

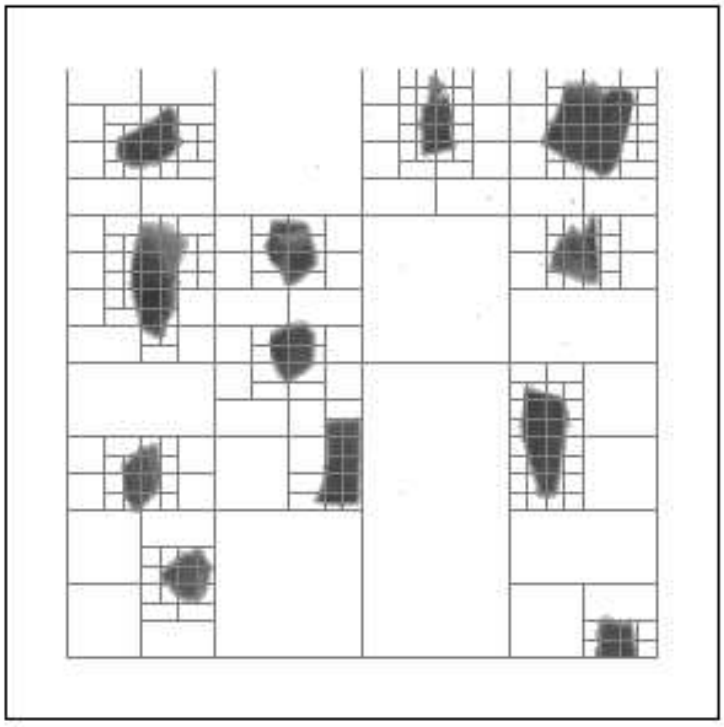


FIG. 5.10 – *Exemple de division récursive d'une image.*

L'utilisation du squelette TF nécessite comme nous l'avons vu l'écriture de trois fonctions :

- une fonction de calcul de prédicat (`predicat`),
- une fonction de traitement des régions homogènes (`calcul`),
- et une fonction de division des régions non homogènes (`division`).

En ce qui nous concerne, le prédicat choisi est donné par l'écart-type des valeurs des pixels composant la région dont on évalue l'homogénéité qui doit être inférieur à un seuil donné. Si c'est le cas alors la fonction `calcul` renvoie une structure de données caractéristique de la région homogène, comprenant :

- les coordonnées et la taille de cette région,
- la moyenne,
- et l'écart-type des pixels qui la composent.

Par contre si la région est déclarée non-homogène, alors la fonction `division` est exécutée. Nous avons choisi de découper la région en deux sous-régions (de même taille).

La spécification fonctionnelle en Caml pour ce programme est la suivante :

```
let image             = lecture_image 512 512
let regions_initiales = division_initiale image
let regions           = tf predicat calcul division
                           accumuler_regions
                           accumulateur_initial
                           regions_initiales

Signature des fonctions (Caml) :
division_initiale :    image
                    -> image list
predicat          :    image
                    -> bool
calcul            :    image
                    -> region
division          :    image
                    -> image list
accumuler_regions :    region
                    -> region list
                    -> region list

Prototypes des fonctions (C) :
void division_initiale( image  *, image  **           );
void          predicat( image   , bool   *            );
void            calcul( image  *, region *            );
void          division( image  *, image  **           );
void accumuler_regions( region *, region **, region ** );
```

Dans cet exemple, une image de 512 pixels par 512 pixels est traitée sur 8 processeurs. La fonction `lecture_image` a pour rôle de fournir l'image à traiter à partir d'une entrée par caméra vidéo par exemple.

Le même phénomène que celui constaté dans pour les deux tests précédent (SCM et DF) sans correction se reproduit ici, pour les mêmes raisons, mais avec une force plus importante dans la mesure où l'algorithme ne réalise aucun calcul dans sa fonction de traitement. Les seules opérations qui sont réalisées sont celles dues au découpage de l'image, et au calcul du prédicat sur les esclaves.

La correction que nous avons apportée, le traitement principal (dans la fonction `compute`) étant inexistant dans cet algorithme, est de répéter une opération arithmétique un très grand nombre de fois au sein de chaque esclave (l'itération est indépendante du nombre d'esclaves mis en jeu).

Le comportement dénoté par les figures 5.11 et 5.12 présente des différences par rapport à ceux que nous avons vus dans les deux tests précédents.

Comme précédemment, le temps d'exécution pour deux processeurs est plus important que celui d'une simple émulation séquentielle. Mais ici la hausse est moins significative. La raison est que le temps d'exécution global de l'algorithme dans sa version corrigée est nettement plus important que dans les tests précédents, alors que le volume des communications reste du même ordre de grandeur. Le temps relevé sur la courbe reflète donc essentiellement le temps de calcul, c'est pourquoi il reste quasi-constant pour 1 et 2 processeurs.

Entre 3 et 5 processeurs, le calcul des prédicats est parallélisé de manière de plus en plus importante avec l'augmentation du nombre de processeurs puisque pour chaque région le calcul du prédicat incombe à un esclave, ce qui diminu le temps d'exécution global.

A partir de 5 processeurs, le temps d'exécution stagne. Ce comportement de convergence rapide est aussi relevé dans les résultats obtenus avec SKiPPER-I (cf. figure 5.13), mais est ici accentué par la méthode de correction des différences de performance entre processseurs mise en œuvre. En effet, la technique utilisée permet de fixer une référence commune en termes de temps d'exécution pour un nombre de processeurs donné. Etant donné qu'on veut corriger les différences de performances entre processeurs, et peu celles entre les réseaux de communication, le choix s'est porté sur une référence commune pour un seul et unique processeur. De plus, pour que la mesure du temps soit fiable, la référence est définie comme le maximum du temps d'exécution (soit ici pour un niveau de division égal à 5 pour SKiPPER-I). Or, dans ce cas, le nombre d'itérations effectuées dansla boucle de correction pour charger le processeur en calculs n'évolue pas avec le nombre de processeurs. Ainsi, à partir de 5 processeurs, le temps de calcul occasionné par la boucle de correction devient prépondérant par rapport aux temps de calcul des prédicats, temps qui deviennent de plus en plus faibles au fur et à mesure que la taille des régions décroît. On constate donc, non pas une convergence, mais une stagnation des résultats due à «l'absorption» du temps de calcul des prédicats par le temps d'itération de la boucle de correction.

L'apparence en «paliers» de la courbe d'exécution (due à une rupture entre 8 et 9 processeurs) n'a pas d'explication, si ce n'est un éventuel changement de taille des données provoquant une diminution brusque du nombre de paquets TCP produits, et donc du temps de communication correspondant.

Contrairement au test avec SKiPPER-I (dont les résultats sont reproduits aux figures 5.13 et 5.14), aucune différence en temps d'exécution ne se fait sentir en fonction du nombre de divisions. Cet état de fait est potentiellement imputable à la technique choisie pour tenir compte des différences de performance entre processeurs qui n'est sans doute pas assez évolutive par rapport à la nature des opérations et au nombre de processeurs dans le cas de figure présent.

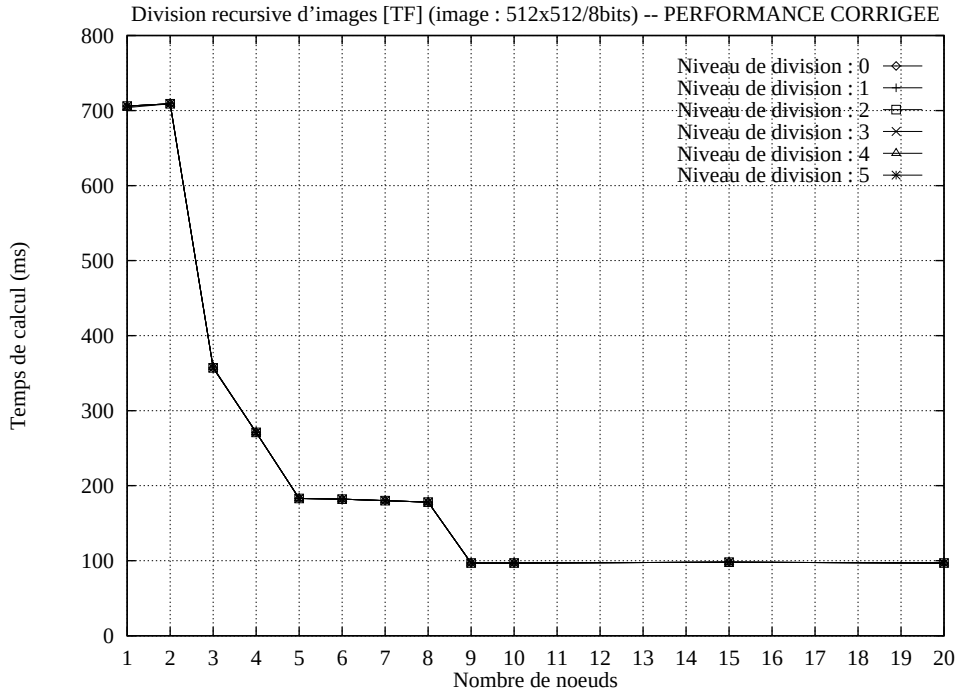


FIG. 5.11 – *Temps d'exécution pour la division récursive d'images (avec correction).*

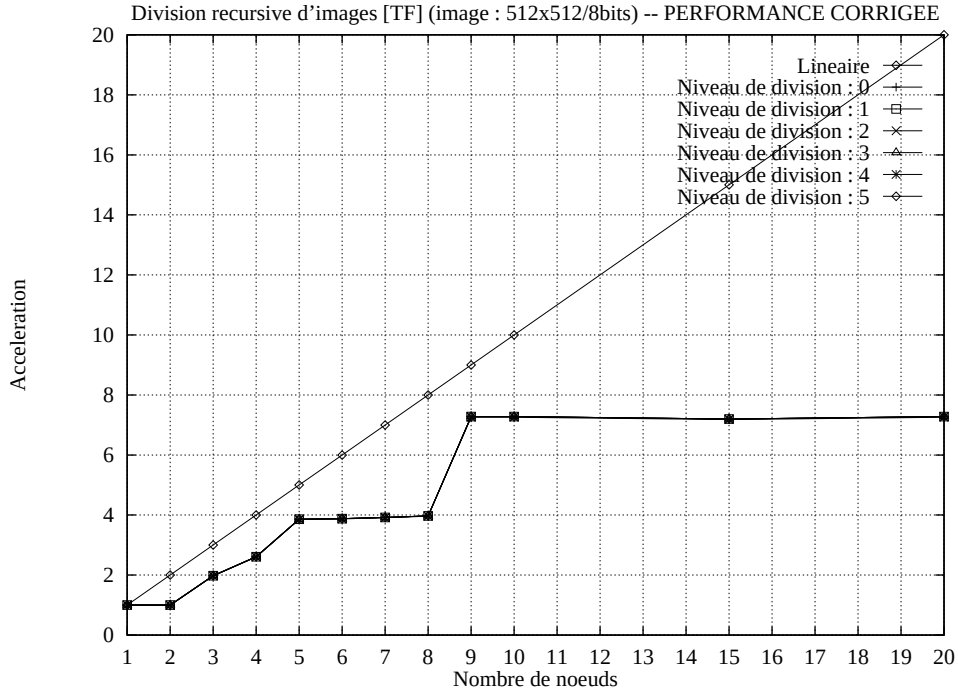


FIG. 5.12 – *Accélération pour la division récursive d'images (avec correction).*

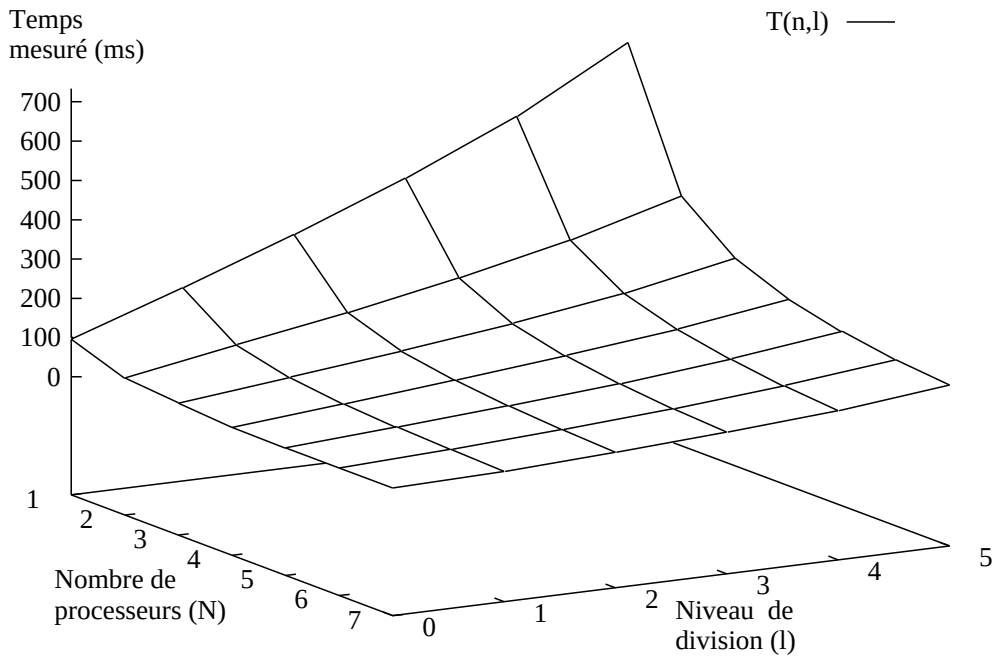


FIG. 5.13 – *Temps d'exécution pour la division récursive d'images avec SKiPPER-I (extrait de [Gin99]).*

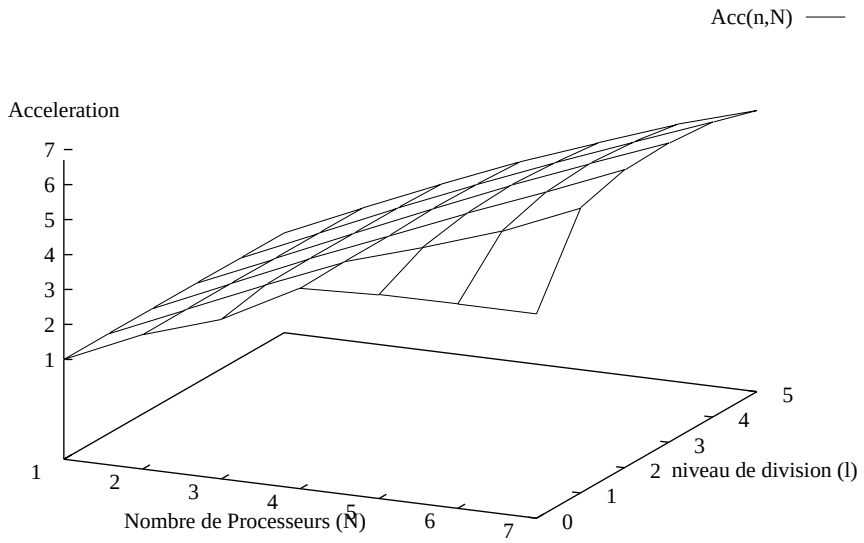


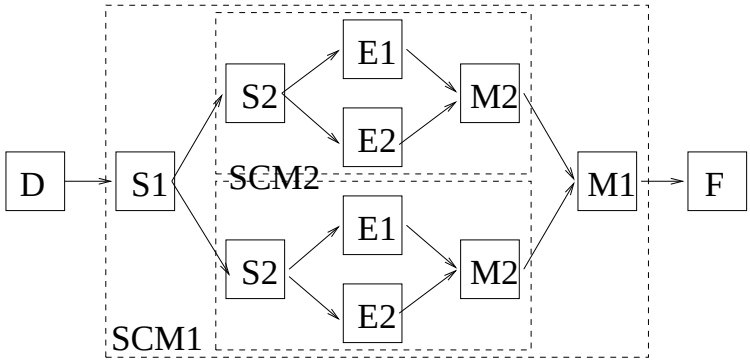
FIG. 5.14 – *Accélération pour la division récursive d'images avec SKiPPER-I (extrait de [Gin99]).*

5.3 Imbrication simple de deux squelettes et comparaison avec une implantation manuelle

Cette section présente une application test simple pour l'évaluation de l'imbrication de deux squelettes avec SKiPPER-II. Une comparaison est faite avec la même application implantée à la main en C et utilisant directement les primitives de communication MPI. L'évaluation a été conduite sur une grappe de 8 stations de travail Sun Ultra 5 interconnectées par un réseau Fast Ethernet 100 Mbits/s (bande passante effective de l'ordre de 8,5 Mo/s par lien).

L'application que nous utilisons ici est volontairement une application «synthétique», par opposition à des applications réelles issues directement du domaine applicatif comme c'est le cas à la section 5.5. L'avantage est sa simplicité qui permet de modifier aisément les paramètres critiques pour notre test que sont la quantité de calculs et le volume des données transmises.

L'application se compose de deux squelettes à parallélisme de données de type SCM dont l'un est imbriqué dans l'autre (voir figure 5.15).



D: fonction d'entree Si: fonctions SPLIT Ei: fonctions esclaves

F: fonction de sortie Mi: fonctions MERGE

⟶ Transferts de donnees

FIG. 5.15 – *Graphe de squelettes de l'application.*

Le squelette SCM englobant est chargé de découper la donnée entrante (un tableau de *N* octets) en deux sous-tableaux de *N/2* octets chacuns. Chaque squelette SCM imbriqué redécoupe les données qui lui arrivent en *n/2* tableaux, où *n* est le nombre total de nœuds de calcul disponibles. Chaque esclave calcule quant à lui *C/n* opérations fbttantes. Les valeurs *N* et *C* peuvent être librement choisies, le rapport *C/N* fixant le ratio calcul/communications de l'application.

Concernant l'implantation à la main qui a été faite de cette application à des fins de comparaison, elle l'a été en mettant «à plat» l'imbrication artificielle (un seul découpage global des données initiales suivies d'un envoi généralisé à tous les esclaves disponibles). L'exécution sur le réseau de stations avec SKiPPER-II a été réalisée en plaçant deux copies du noyau par nœud de calcul.

Nous avons pris pour les tests les trois configurations suivantes :

- C=100 MFlops, N=1 Mo,
- C=100 MFlops, N=10 Mo,
- C=10 MFlops, N=10 Mo.

Les figures 5.16, 5.17 et 5.18 présentent les résultats que nous avons obtenus.

La figure 5.16 montre un cas dans lequel le rapport calcul/communication permet de se rapprocher d'une accélération linéaire. Dans ce cas, les performances de SKiPPER-II et de la version C/MPI sont d'ailleurs très proches. On peut constater ici que le surcoût exécutif («overhead») occasionné par SKiPPER-II est inférieur à 10 %. Il est essentiellement dû à deux facteurs. Le premier est la gestion centralisée des ressources libres faite par le noyau (voir à ce propos les sections 4.3.4.1 page 103, et 4.6.2 page 113). Le second est le lancement des processus maîtres des squelettes imbriqués. L'influence des coûts de communication mis en jeu dans le schéma d'imbrication apparait clairement sur la figure 5.17. Dans cette hypothèse les communications supplémentaires entre les processus du squelette englobant et les processus du squelette imbriqué, par rapport à une version sans imbrication, peuvent devenir rédibitoires. En effet, l'utilisation d'une imbrication entraîne ici la circulation d'une quantité double de données : un premier découpage par le squelette englobant implique une première communication de toutes les données initiales, et un second découpage au niveau du squelette imbriqué en occasionne une autre. C'est ce qu'on peut constater sur la figure 5.18 qui montre le cas d'un rapport calcul/communications très bas. Enfin, ce qui est intéressant de retenir de cette expérimentation est le comportement de SKiPPER-II avec l'augmentation du nombre de nœuds de calcul : il suit celui de la version parallélisée à la main, et cela d'autant plus près que le nombre de nœuds est important. Ainsi, le comportement du prototype reproduit-il fidèlement celui que l'utilisateur pourrait attendre de son application parallélisée.

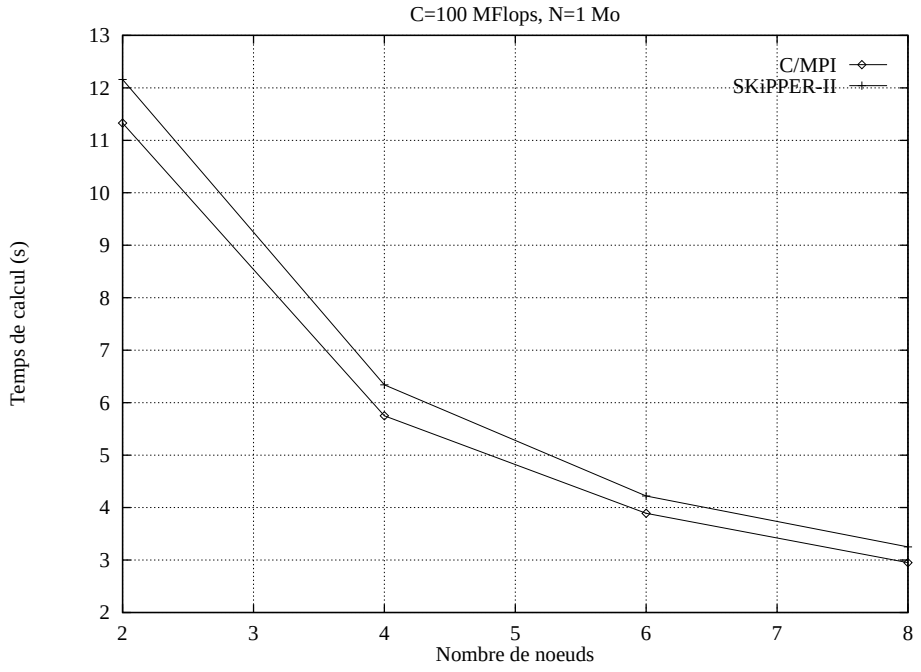


FIG. 5.16 – *Comparaison d'une implantation sous SKiPPER-II et en C/MPI de deux SCM imbriqués (C=100 MFlops/N=1 Mo).*

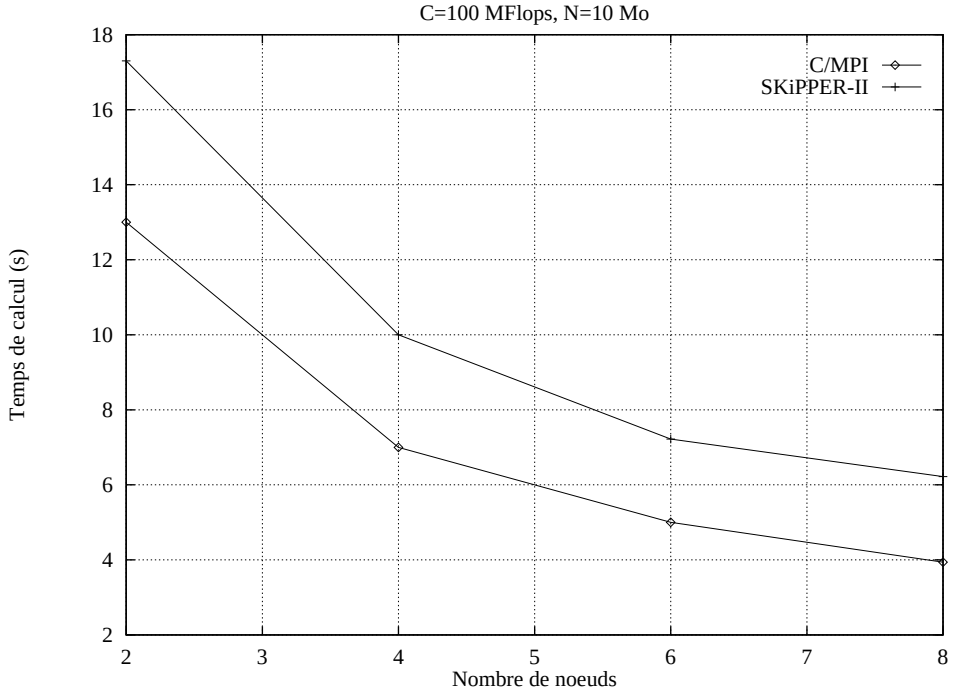


FIG. 5.17 – *Comparaison d'une implantation sous SKiPPER-II et en C/MPI de deux SCM imbriqués (C=100 MFlops/N=10 Mo).*

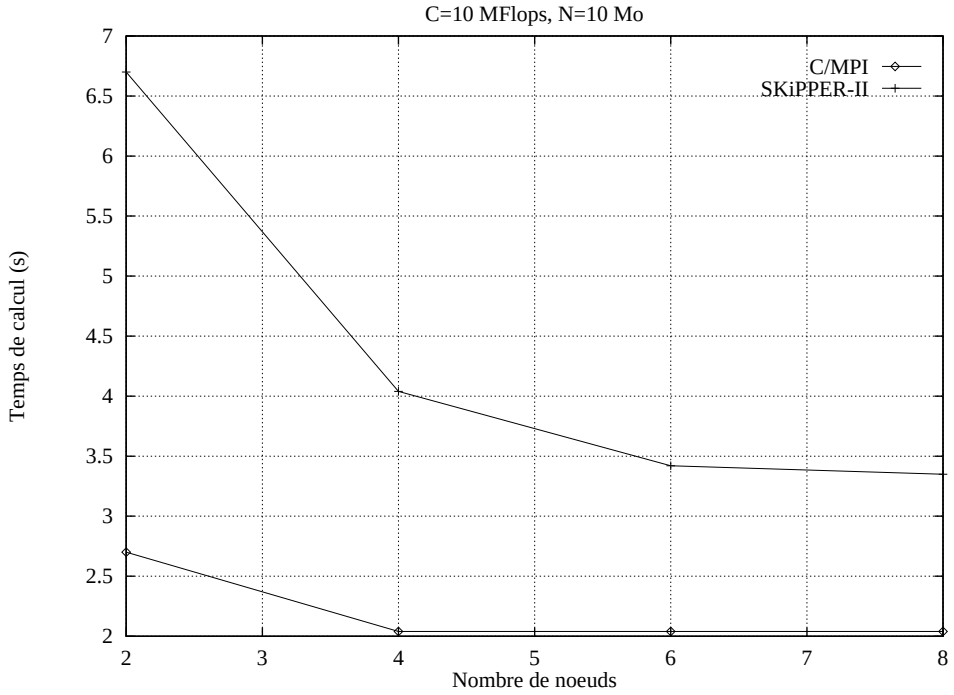


FIG. 5.18 – *Comparaison d'une implantation sous SKiPPER-II et en C/MPI de deux SCM imbriqués (C=10 MFlops/N=10 Mo).*

5.4 Multiplication de deux matrices d'entiers de longueur arbitraire par imbrication de deux squelettes

5.4.1 Présentation de l'algorithme

L'application présentée ici est le calcul du produit de deux matrices dont les éléments sont des entiers de longueur arbitraire, c'est-à-dire pouvant être formés d'un nombre quelconque de chiffres.

L'implantation [5] qui en a été faite exploite deux squelettes à parallélisme de données de type SCM bien adaptés aux calculs du fait de leur grande régularité : volume des données connu et traitements indépendant du contenu. Les deux squelettes sont imbriqués. Le premier (le squelette englobant) découpe l'une des matrices en lignes (en blocs de lignes plus exactement, le nombre de lignes par bloc dépendant du nombre de processeurs disponibles) et demande à chacun de ses esclaves (constitués par le second SCM, squelette imbriqué) d'effectuer le calcul matrice-vecteur. Ce dernier découpe donc la seconde matrice en colonnes (là encore en blocs de colonnes dont la taille dépend du nombre de processeurs) et fait faire le produit vecteur-vecteur par chacun de ses propres esclaves. Dans cet exemple il est clair que l'imbrication n'est pas une nécessité pour faire le calcul [CMT94]. Si elle a été mise en œuvre c'est pour évaluer le comportement de SKiPPER-II sur un programme type déjà exploité par d'autres concepteurs d'environnements parallèles fondés sur les squelettes et autorisant l'imbrication [MSBK00] [SMH01]. Il nous a donc paru intéressant de renouveler l'expérience avec SKiPPER-II. Cela étant, une version n'exploitant qu'un seul et unique SCM a aussi été produite. Les résultats obtenus sont très proches de la version imbriquée. Cela est dû à ce que la version imbriquée ne fait rien d'autre que de réaliser le découpage des matrices en deux temps (un découpage de matrice par niveau d'imbrication), ce que la version non imbriquée fait directement dans son ensemble au niveau de la fonction *split* du seul SCM existant. La différence de temps d'exécution se limite donc au temps mis à distribuer correctement les données jusqu'aux esclaves chargés des traitements, temps qui est plus important dans la version imbriquée que dans la version non imbriquée.

5.4.2 Résultats obtenus pour deux dimensions de matrices en faisant varier la taille des entiers

Deux dimensions de matrices ont été utilisées : 10x10 et 25x25. Pour chaque dimension, on a fait varier le nombre de chiffres par entier composant les matrices entre 10 et 25. L'intérêt de faire varier la dimension des nombres composants les matrices est de modifier la quantité de calculs effectués par les esclaves du squelette imbriqué sans pour autant modifier la quantité d'opérations (mutliplications) à effectuer au niveau du calcul matriciel. En effet, dans ce cas de figure le volume des données à communiquer augmente linéairement avec le nombre de chiffres par entier, alors que le volume des calculs pour multiplier deux entiers a une croissance quadratique [6].

L'ensemble des tests ont été menés sur une machine Beowulf équipée de 32 nœuds de calcul de type Intel Celeron cadencés à 533 MHz. Le Beowulf était équipé d'un réseau commuté Fast Ethernet à 100 Mbits/s. Les résultats sont présentés dans les figures 5.19 à 5.22.

5. En annexe D sont propos´es des extraits des fi chiers sources de cette application.

6. Le ratio calcul/communication varie donc.

D'une manière générale ces courbes, quelle que soit la dimension des matrices ou la taille des entiers qui les composent montrent les phases suivantes dans le fonctionnement de SKiPPER-II.

La première apparait lorsque seulement 2 processeurs sont utilisés. A ce moment-là on constate que le temps d'exécution est plus important que lorsque seulement un processeur est utilisé. Le phénomène est moins important avec une dimension de matrices plus importante. Il s'explique parce qu'avec l'utilisation de 2 processeurs le squelette SCM englobant est déployé sur les unités de calcul. De ce fait, et conformément à ce qui a été présenté au chapitre 4, utiliser 2 nœuds de calcul au lieu d'un seul ne fournit pas plus de puissance de calcul, mais par contre accroît très sensiblement le volume des communications. Cela est dû au fait qu'un des deux nœuds de calcul est utilisé comme processus de répartition des traitements à effectuer et non pour réaliser concrètement des traitements. Avec peu de processeurs on peut noter que le gain en performances est beaucoup plus important lorsqu'on passe de 2 à 3 processeurs avec un volume de calcul plus important (25 chiffres plutôt que 10).

Ensuite le nombre de processeurs fait baisser le temps d'exécution. Dans cette phase le parallélisme potentiel de l'application est exploité au mieux du nombre de processeurs : plus le nombre de processeurs est grand, plus l'application est rapide. En effet la finesse du découpage des matrices, et donc le nombre d'opérations qui sont faites en parallèle, dépend directement du nombre de processeurs disponible au moment du découpage (donc en cours d'exécution).

A la fin de cette deuxième phase (variable en fonction du volume des calculs à réaliser) s'en amorce une troisième dans laquelle augmenter le nombre de processeurs conduit à faire chûter les performances. L'explication vient ici du fait, qu'à partir d'un certain seuil, il est devient coûteux de paralléliser les calculs. En effet, le grain de parallélisation devient alors trop fin (par exemple atteindre le calcul d'une simple opération arithmétique entre deux entiers). Le temps mis pour réaliser une opération «élémentaire» étant très faible par rapport au temps de communication des données que l'opération requiert, les performances vont chûter : plus de communications individuelles vont être amorçées, alors que le temps de calcul sur chaque processeur aura déjà atteint sa limite inférieure.

On peut aussi noter que les résultats montrent une convergence des temps d'exécution vers la même limite, quelque soit le nombre de chiffres utilisé pour former un entier, et ce pour une dimension de matrice donnée. Cette convergence s'explique par le fait qu'avec plus de processeurs que nécessaire pour effectuer toutes les opérations «élémentaires» (entre entiers) la puissance supplémentaire ne peut plus être exploitée et les temps d'exécution vont stagner. En effet, le nombre d'opérations élémentaires à réaliser pour une dimension de matrice donnée est fixe. Si le nombre de processeurs mis en œuvre est plus important, les processeurs en surnombre ne seront pas utilisés, et ainsi le temps d'exécution ne changera pas.

Au niveau de l'accélération on pourra noter que, lorsque le volume des calculs est suffisamment important (plus de 20 chiffres par entier), l'accélération relative $\Delta accélération / \Delta nombre\ de\ processseurs$ est quasi-linéaire, ne s'écartant que d'un facteur fixe de l'accélération idéale. Mais le comportement du noyau lors de la première phase que nous avons mentionnée introduit un «retard» dans la courbe d'accélération en décalant le point où l'accélération devient supérieure à 1 de 2 vers 3 processeurs.

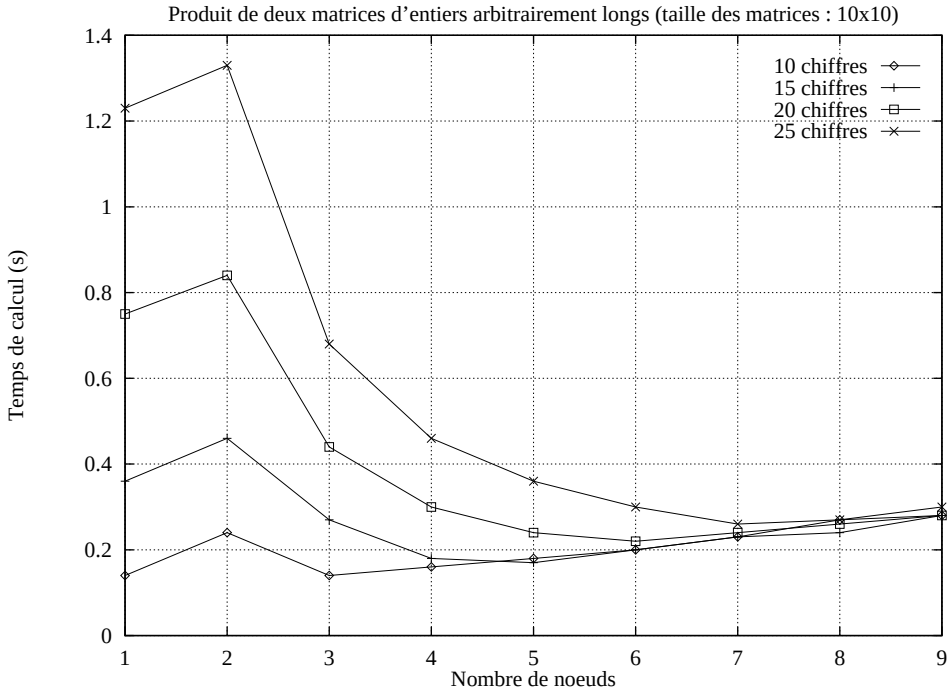


FIG. 5.19 – *Produit de deux matrices 10x10 d'entiers de longueurs arbitraires (temps d'exécution).*

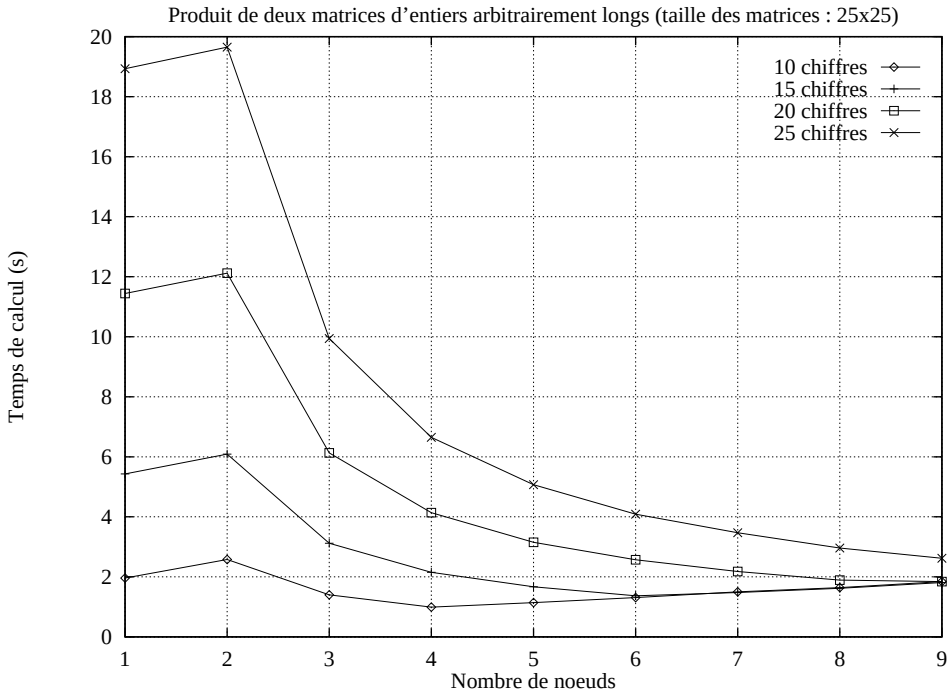


FIG. 5.20 – *Produit de deux matrices 25x25 d'entiers de longueurs arbitraires (temps d'exécution).*

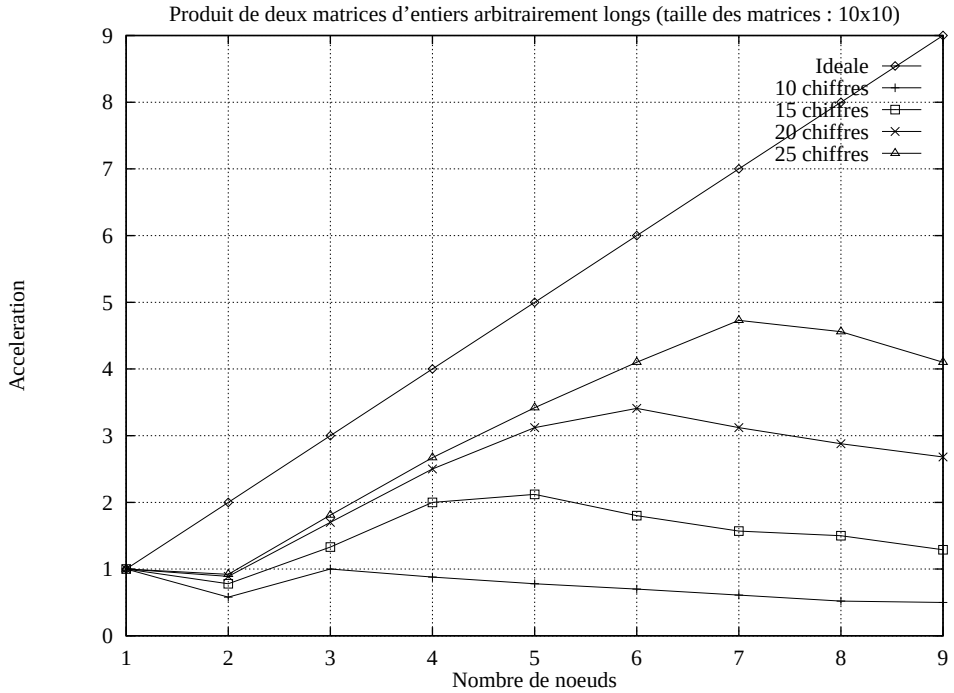


FIG. 5.21 – *Produit de deux matrices 10x10 d'entiers de longueurs arbitraires (accélération).*

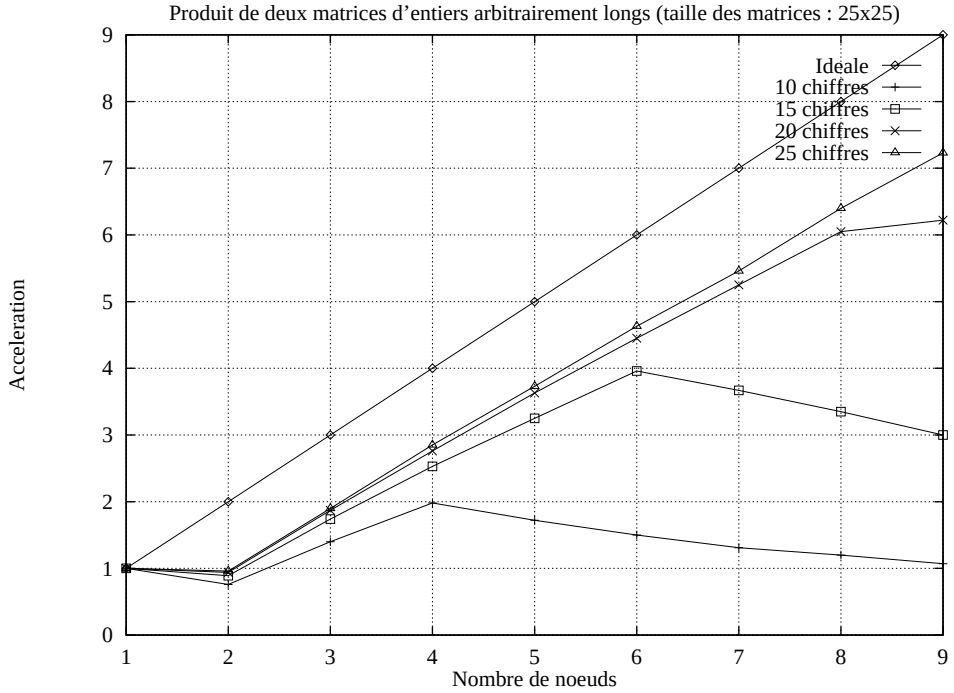


FIG. 5.22 – *Produit de deux matrices 25x25 d'entiers de longueurs arbitraires (accélération).*

5.4.3 Résultats obtenus pour des entiers longs (50 chiffres)

De la même manière nous avons fait un test avec des entiers de 50 chiffres sur des matrices 10x10 et 25x25. Les résultats sont donnés aux figures 5.23 et 5.24. Ils montrent ici une meilleure accélération, les données étant plus longues à traiter.

5.4.4 Comparaison des résultats obtenus avec ceux de Michaelson et coll.

Dans [SMH01] Michaelson et coll. ont testé leur environnement de développement (voir la section 1.5.3.2 page 50) sur la même application et sur la même machine. Nous avons d'ailleurs développé et évalué ce programme de test d'après ces travaux.

La comparaison de notre expérimentation avec celle rapportée dans [SMH01] fait apparaître une accélération relative quasi-linéaire pour un nombre plus important de processeurs avec SKiPPER-II (la linéarité s'arrête vers 10 processeurs avec l'environnement de Michaelson et coll. pour une matrice 25x25 à entiers de 25 chiffres). On notera que dans [SMH01] l'imbrication des deux squelettes n'est activée qu'à partir de 7 processeurs et que pour 2 processeurs l'exécution des squelettes est maintenue séquentielle [MSBK00]. Sur ce dernier point SKiPPER-II se rapproche de ce fonctionnement en ne déployant les processus émulant les squelettes qu'au fur et à mesure de la disponibilité des ressources. Par contre l'imbrication est exploitée dès le plus petit nombre de processeurs disponibles.

Il est intéressant de noter la différence de comportement entre SKiPPER-II et l'environnement décrit dans [SMH01] lorsque le nombre de processeurs croît. Avec SKiPPER-II, l'accélération se maintient jusqu'à un nombre de processeurs optimum dépendant de la quantité de calculs à effectuer par rapport au volume de données à transmettre. Ensuite la tendance s'inverse définitivement. Alors que pour l'environnement présenté dans [SMH01] l'accélération est moins régulière : d'abord linéaire, elle stagne ensuite pour reprendre ; sa progression se fait essentiellement par paliers. Avec SKiPPER-II, les courbes de performance montrent un comportement régulier qui, contrairement à cette autre approche de l'imbrication, conduisent à une chûte régulière de l'accélération lorsque cette chûte s'amorce ; l'autre cas présente pour certaines applications des reprises de l'accélération après un début de chûte des résultats.

Enfin, conformément à notre analyse, Michaelson et coll. rapporte dans [SMH01] que la version imbriquée de cet algorithme n'a d'intérêt que pour l'évaluation de leur environnement de parallélisation. De plus, comme avec SKiPPER-II, la version non imbriquée et la version imbriquée de ce programme présentent des performances très proches.

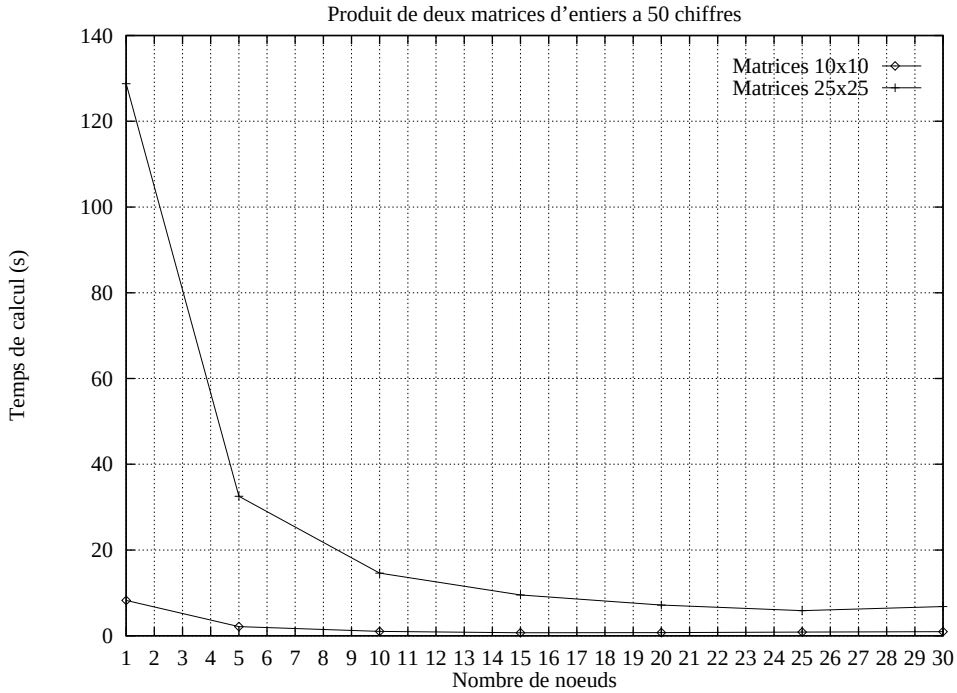


FIG. 5.23 – *Produit de deux matrices d'entiers de 50 chiffres (temps d'exécution).*

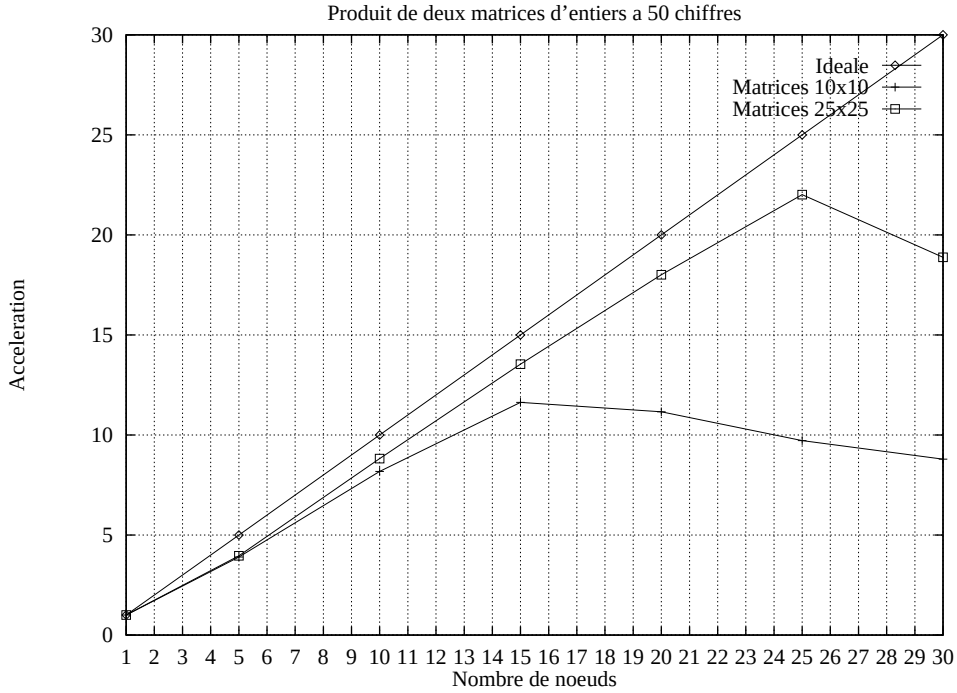


FIG. 5.24 – *Produit de deux matrices d'entiers de 50 chiffres (accélération).*

5.5 Cas réel d'imbrication : algorithme de suivi de visages

Cette section présente la parallélisation d'une application complète de suivi de visages. L'algorithme a été développé au sein de notre laboratoire par une équipe proposant des algorithmes de vision artificielle séquentiels. Aucune intention de parallélisation immédiate ou future le concernant n'avait été supposée au moment de sa conception. De ce fait, il a été intéressant de se confronter au cas réel de la parallélisation d'une application non initialement prévue pour être parallélisée[7].

L'algorithme de suivi de visage a aussi – et surtout – été retenu pour montrer l'utilité de disposer de possibilités d'imbrication de squelettes dans un cas réel, chose qui, à notre connaissance, a rarement été mise en évidence notamment dans le cadre spécifique de la vision artificielle.

5.5.1 Présentation de l'algorithme de suivi de visages

5.5.1.1 Introduction

L'application utilisée ici pour mettre en lumière l'intérêt de disposer de la possibilité d'imbriquer des squelettes dans les cas applicatifs réels est un algorithme 3D de suivi de visages humains dans des séquences d'images. Le suivi se fonde uniquement sur l'apparence générale des visages dans les images vidéo (*i.e.* par mise en correspondance avec une base de données de différentes vues des visages recherchés). Il reprend un algorithme précédemment développé [JD01] pour le suivi de mouvements 2D d'un motif dans un fbt vidéo. Dans le champ applicatif de cet algorithme se trouve notamment la visioconférence, très gourmande en bande passante, et pour laquelle son application peut réduire le volume des données transférées.

Dans l'approche 3D utilisée ici un visage est représenté par une collection d'images 2D appelées *vues de référence*. Ce sont ces vues qui, une fois repérées dans l'image, vont être suivies au cours de leur déplacement. On définit également un *motif* comme étant une région d'intérêt dans l'image actuellement présentée au traitement. Son échantillonnage selon une procédure bien précise [DDJ01] fournit un vecteur de niveaux de gris utilisé pour la mise en correspondance du motif courant avec un motif de référence de la base.

La technique de suivi mise en œuvre fait intervenir deux étapes différentes.

La première est une étape d'apprentissage. Elle se déroule «hors-ligne», c'est-à-dire notamment sans contraintes critiques de temps *a priori*. Elle est destinée à construire la base des vues de référence et des informations qui leur sont associées. Pour chaque vue une *matrice d'interaction* (notée A) est calculée. Cette matrice fait le lien entre la différence de gris existant entre le motif de référence suivi et le motif en cours d'échantillonnage d'une part, et le mouvement dit «fronto-parallèle»[8] de ce dernier d'autre part. L'aspect global du motif identifié comme étant le visage à suivre n'est pas modifié par ce type de mouvement. Cependant la position, l'orientation et la taille du motif peuvent changer.

La seconde étape est indépendante de la première. Elle est exécutée «en-ligne» et consiste à prédire la situation du visage à suivre dans l'image courante (en termes de position, orientation et taille) (cf. figure 5.25).

7. Et encore moins par l'environnement SKiPPER.

8. La d´efi nition de mouvement «fronto-parall`ele» donn´ee dans [DD]est celle d'un mouvement du visage dans un plan parall`ele au plan de l'image lui-mˆeme.

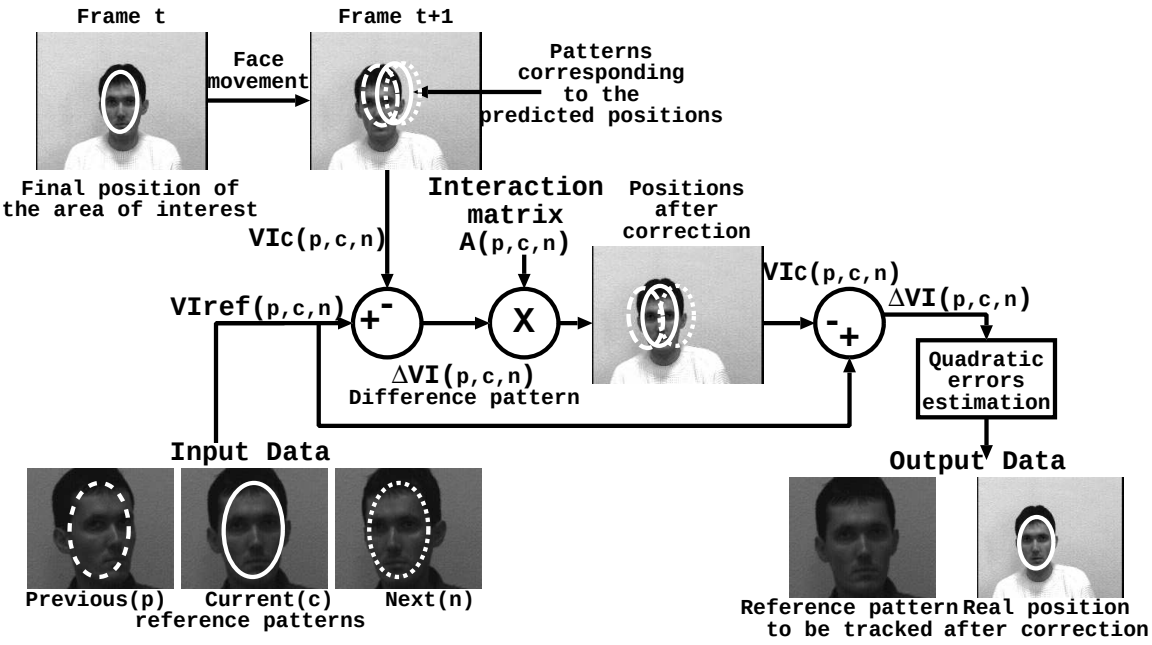


FIG. 5.25 – *Synoptique de l'algorithme de suivi de visages (extrait de [CDS+]).*

La prédiction est faite dans une zone elliptique. Pour le motif de référence actuellement suivi et ses plus proches voisins dans la collection d'images de référence, on calcule la correction associée à apporter aux paramètres de l'ellipse. Chaque correction est obtenue en multipliant la différence de niveaux de gris existant entre le motif contenu dans la zone elliptique et le motif de référence en cours de traitement par la matrice d'interaction associée. On obtient alors une nouvelle position pour la zone elliptique qui est supposée se superposer parfaitement avec le visage à suivre dans l'image. On peut alors estimer l'erreur quadratique de la différence de niveaux de gris ΔVI entre le motif contenu dans la zone d'intérêt VIc et le motif de référence $VIref$. On retient alors le motif de référence donnant la plus petite erreur quadratique comme nouveau motif de référence pour établir la position dans l'image suivante. L'utilisation simultanée de plusieurs motifs de référence permet de gérer les variations d'aspect du visage lors de mouvements en rotation de la tête du sujet. En outre elle permet de changer de motif de référence au cours du processus de suivi sans pour autant arrêter ce dernier. Comme la fréquence du traitement est importante par rapport à la vitesse de mouvement du visage à suivre il n'est pas nécessaire d'utiliser un algorithme de prédiction (les variations de position du motif restent du même ordre de grandeur que celles apprises lors de la phase d'apprentissage «hors-ligne»).

5.5.1.2 Modélisation de l'apparence de visages dans une image

Les visages sont des objets extrêmement variables et déformables induisant des aspects très différents dans une image en fonction de la pose, de l'éclairement, de l'expression et de la personne elle-même. Dans l'approche 3D retenue pour l'algorithme de suivi, un visage est représenté par une collection d'images (en 2D). Chacune de ces images représente elle-même l'un des motifs de référence utilisés pour rendre compte d'une position particulière du visage par rapport à la caméra. C'est cette base qui rend le suivi possible. L'acquisition de vues intermédiaires permet de mémoriser durant la phase d'apprentissage des visages la zone elliptique résultant du suivi pour les motifs de référence les plus proches dans la collection d'images (cf.

figure 5.26).

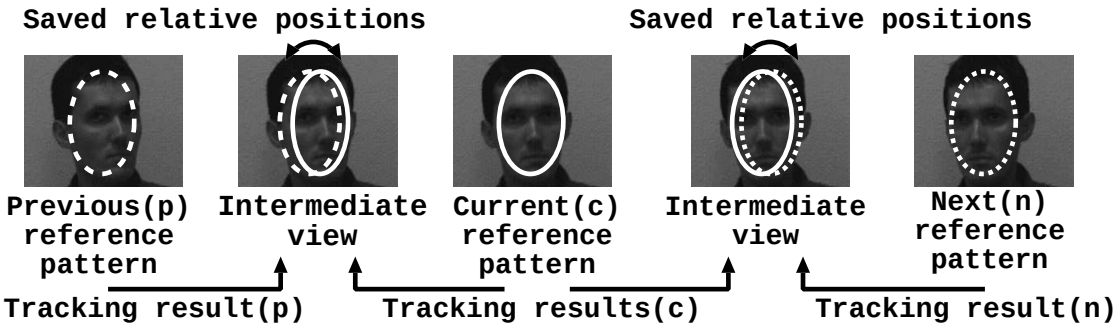


FIG. 5.26 – *Vues intermédiaires utilisées pour la construction de la base (extrait de [CDS+]).*

De ce fait, durant la phase de suivi, il est possible de calculer la position relative des ellipses des motifs de référence les plus proches dans la collection d'images par rapport à la position courante de l'ellipse dans l'image correspondant au motif de référence actuellement suivi avant de calculer la correction des paramètres. L'évaluation simultanée de trois motifs de référence permet de prendre en compte des variations d'aspect du motif suivi. L'analyse des motifs se faisant simultanément elle évite aussi d'interrompre le processus de suivi (cf. figure 5.27).

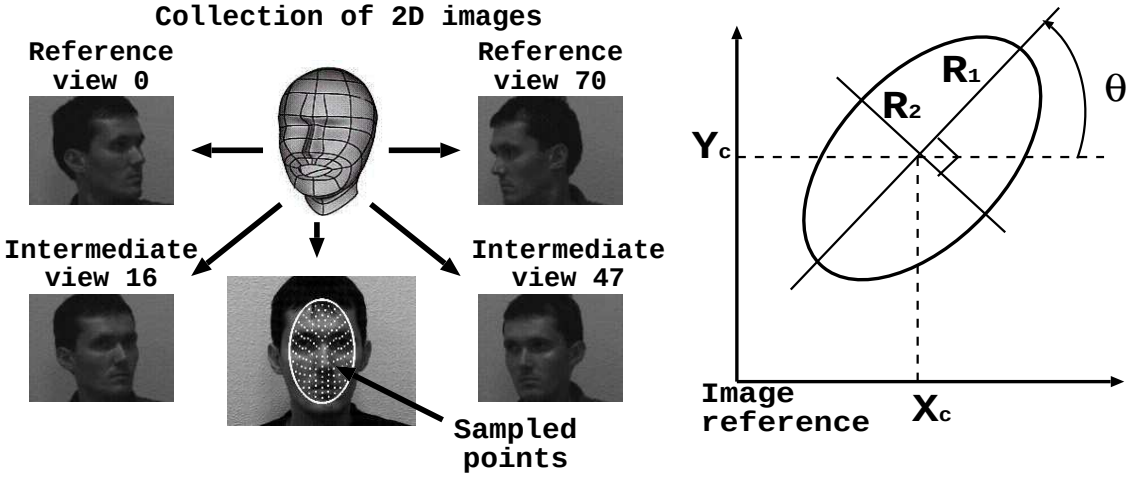


FIG. 5.27 – *Modèle de visage et échantillonnage d'un motif (extrait de [CDS+]).*

La base de donnée construite lors de la première étape de l'algorithme incorpore 71 vues (8 motifs de référence et 63 vues intermédiaires) afin de gérer tout mouvement en rotation de la tête du sujet sur un domaine couvrant 180 degrés. Le motif à suivre est représenté par un vecteur de forme (vecteur de niveaux de gris de dimension N qui est le nombre de points d'échantillonnage pris dans la région d'intérêt). Cette représentation est choisie pour être invariante en position, orientation et changement d'échelle du motif. C'est la raison pour laquelle la forme de la zone d'intérêt a été choisie elliptique. Les différents points d'échantillonnage (points blancs sur la figure 5.27) sont distribués sur un ensemble de 10 ellipses concentriques allant de la plus petite à la plus grande. Cette manière d'échantillonner, qui présente une disposition uniforme des points, permet de limiter l'influence que peuvent avoir les changements d'expression du visage lors du suivi.

La position et la forme de l'ellipse sont définies par 5 paramètres constituants un vecteur (cf. figure 5.27) :

- les coordonnées du centre de l'ellipse (X_c, Y_c),
- son orientation (θ),
- et les longueurs de ses grand et petit axes (R_1, R_2) ($R_2 = k * R_1$, où k est un facteur d'échelle fixé par l'utilisateur).

Une fois obtenu, le vecteur de niveaux de gris est centré et normalisé afin de limiter l'influence des variations lumineuses de la scène.

5.5.1.3 Principe du suivi de visages et interprétation géométrique

Comme nous venons de le voir, le motif suivi est confiné dans une ellipse dont la forme et la position dans l'image analysée sont données par le vecteur de paramètres μ de taille p (ici $p = 4$) et $\mu = (X_c, Y_c, R_1, \theta)^t$ avec $R_2 = k * R_1$. Si on note μ_r le vecteur de paramètres correspondant à la position réelle du motif et μ_p celui de la position prédite, alors on évalue simplement leur écart par la différence : $\Delta\mu = \mu_r - \mu_p$. Le motif contenu dans l'ellipse prédite est échantillonné pour obtenir le vecteur de forme courant VI_c de dimension N. Le vecteur de forme du motif de référence suivi est quant à lui noté VI_{ref}, d'où la différence $\Delta VI = VI_{ref} - VI_c$. En remarquant que $\Delta\mu = A\Delta VI$ on constate que le problème du suivi peut être formulé comme la détermination du vecteur $\Delta\mu$ en supposant que les variations en position d'un visage dans l'image correspondent aux variations des paramètres d'une simple transformation géométrique (cf. figure 5.28). Dans le cas de l'algorithme qui est présenté, c'est une transformation affine rigide qui est utilisée dans laquelle les paramètres de l'ellipse sont aussi les paramètres de la transformation géométrique. Finalement la matrice A est une matrice d'interaction de dimension $(p \times N)$ correspondant au calcul de la relation linéaire entre un ensemble de différences de niveaux de gris ΔVI et une correction $\Delta\mu$ des paramètres du vecteur μ (calcul fait pendant la phase «hors-ligne» de l'algorithme).

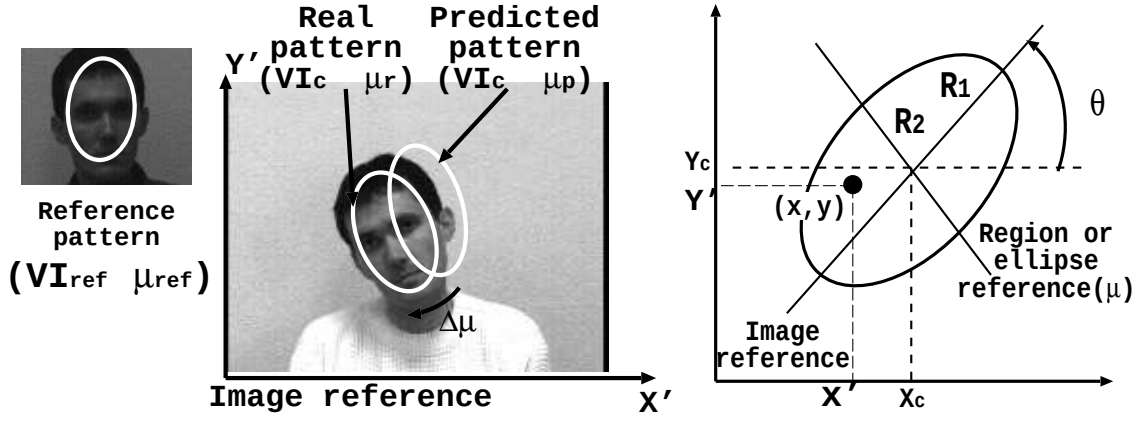


FIG. 5.28 – *Principe du suivi de visage (extrait de [CDS+]).*

5.5.1.4 Détermination de la matrice d'interaction A pour un motif de référence donné

Le calcul de la matrice d'interaction A est réalisé durant la phase «hors-ligne» de l'algorithme (lors de la construction de la base de données). Pour ce faire, et contrairement à d'autres algorithmes, la méthode proposée n'utilise pas de matrices Jacobiennes pour les vues de références. La matrice A est estimée par une minimisation au sens des moindres carrés exploitant une technique fondée sur une décomposition en valeurs singulières. De cette manière la convergence est améliorée [JD01].

L'obtention de cette matrice rend possible la mise à jour des paramètres de la transformation affine rigide pendant la phase de suivi proprement dite. Les étapes du suivi sont alors les suivantes. Tout d'abord une ellipse est appliquée manuellement par un opérateur humain sur le motif de référence. Ce dernier est alors echantillonné pour obtenir le vecteur de forme VI_{ref}. Cette initialisation permet aussi de fixer un ratio (la valeur du paramètre k) entre les deux rayons de l'ellipse ($k = R_2/R_1$). La position de l'ellipse peut alors être perturbée autour de cette position de référence (le paramètre k est laissé constant). La figure 5.29 illustre cela.

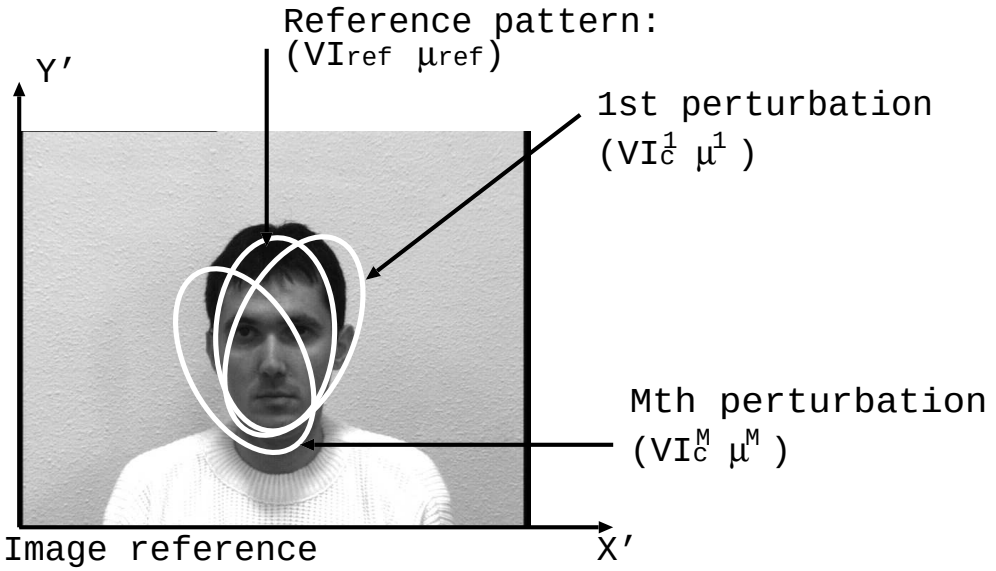


FIG. 5.29 – *Calcul de la matrice d'interaction A (extrait de [CDS+]).*

Pour chaque perturbation i, la variation des paramètres de la transformation $\Delta\mu^i = (\Delta X_c^i, \Delta Y_c^i, \Delta R_1^i, \Delta\theta^i)^t$ tout comme les valeurs du vecteur correspondant au motif échantillonné dans l'ellipse VI_c^i sont mémorisés. A partir de là, il est possible d'estimer la matrice A à partir de M mesures du type précédent pour N points échantilonnés, à la condition que $M \geq N$. Pratiquement cela conduit à 500 perturbations de paramètres et un échantillonnage de l'ordre de 170 points. Par conséquent il faut résoudre un système sur-déterminé de M équations à N inconnues pour chaque paramètre de la transformation (soit 4 dans le cas présent). En fait la résolution d'un seul système linéaire, où plus exactement le calcul d'une seule matrice pseudo-inverse, est nécessaire. En effet si on note $\Delta VI^j = (\Delta i_1^j, \Delta i_2^j, \ldots, \Delta i_N^j)^t$ la différence vectorielle entre le motif de référence VI_{ref} et le motif correspondant à la $j^{ème}$ perturbation VI_c^j, et $A = (AX_c, AY_c, AR_1, A\theta)^t$ la matrice d'interaction, on peut obtenir la ligne $A\theta$ de cette dernière comme suit (cette ligne est relative à l'orientation de l'ellipse) :

$$\begin{pmatrix} \Delta i_1^1 & \dots & \Delta i_N^1 \\ \vdots & \dots & \vdots \\ \Delta i_1^M & \dots & \Delta i_N^M \end{pmatrix} \begin{pmatrix} A\theta_1 \\ \vdots \\ A\theta_N \end{pmatrix} = \begin{pmatrix} \Delta\theta^1 \\ \vdots \\ \Delta\theta^M \end{pmatrix}$$

Ce qui peut aussi être écrit de manière plus condensée :

$$M_{\Delta VI} * A\theta = \Delta\theta$$

La solution finale est donc :

$$\begin{cases} A\theta = (M_{\Delta VI}^t M_{\Delta VI})^{-1} M_{\Delta VI}^t \Delta\theta = M_{\Delta VI}^+ \Delta\theta \\ AX_c = M_{\Delta VI}^+ \Delta X_c \\ AY_c = M_{\Delta VI}^+ \Delta Y_c \\ AR_1 = M_{\Delta VI}^+ \Delta R_1 \end{cases}$$

La matrice $M_{\Delta VI}^+$ est la matrice pseudo-inverse de $M_{\Delta VI}$.

5.5.1.5 Basculement entre deux vues de référence

Plusieurs motifs de référence (en provenance de la base) peuvent être suivis en effectuant un basculement entre les différentes vues mémorisées dans la base d'images. Pour ce faire, une comparaison régulière de l'erreur quadratique obtennue à l'issue du suivi entre le motif de référence actuel et ses plus proches voisins dans la base est établie. Le motif de référence donnant la plus petite erreur est considéré comme le motif à suivre dans la prochaine image. Cependant cela ne peut se faire directement. Pendant la phase «hors-ligne» sont calculées pour les vues intermédiaires, positionnées entre deux motifs de référence consécutifs dans la base, les différentes positions de l'ellipse correspondant au résultat du suivi pour chacun des motifs de référence (cf. figure 5.26). Ces vues intermédiaires sont notamment choisies parce qu'il est supposé que le changement de motif de référence pendant la phase de suivi proprement dite ne survient qu'autour de ces vues. Pendant la phase de suivi donc, ces différents résultats sont exploités avec les paramètres évalués pour l'ellipse associée au motif en cours de suivi pour calculer les positions à prédire du visage concernant les deux motifs précédent et suivant dans l'image courante avant les corrections et estimations de l'erreur quadratique qui leur est associées.

5.5.2 Parallélisation de l'algorithme de suivi avec imbrication de squelettes

5.5.2.1 Principe

A partir de maintenant nous ne considérons plus que la seconde étape de l'algorithme, c'est-à-dire la phase de suivi proprement dite (celle «en-ligne»). En effet, c'est elle que nous avons choisie pour la parallélisation. Nous en proposons donc la parallélisation suivante :

- Les calculs sur les motifs de référence précédent, actuel et suivant peuvent être clairement réalisés de manière complètement indépendante les uns des autres, et donc en parallèle (voir la section 5.5.1.2 concernant la notion de motif de référence). De plus, ces traitements représentent la même charge de travail pour chaque processeur s'ils sont faits

indépendamment les uns des autres. Nous considérons donc que le premier niveau de parallélisation coïncide avec un premier squelette à parallélisme de données (squelette noté A sur la figure 5.30). Ce squelette est utilisé pour réaliser la comparaison entre les trois motifs de référence calculés en parallèle.

- L'étape de multiplication matricielle incluse dans le traitement de chacun des motifs de référence peut aussi être parallélisée. Ce second niveau de parallélisation correspond aussi à un squelette à parallélisme de données mais qui est imbriqué dans le précédent (il est représenté par le squelette B sur la figure 5.30).

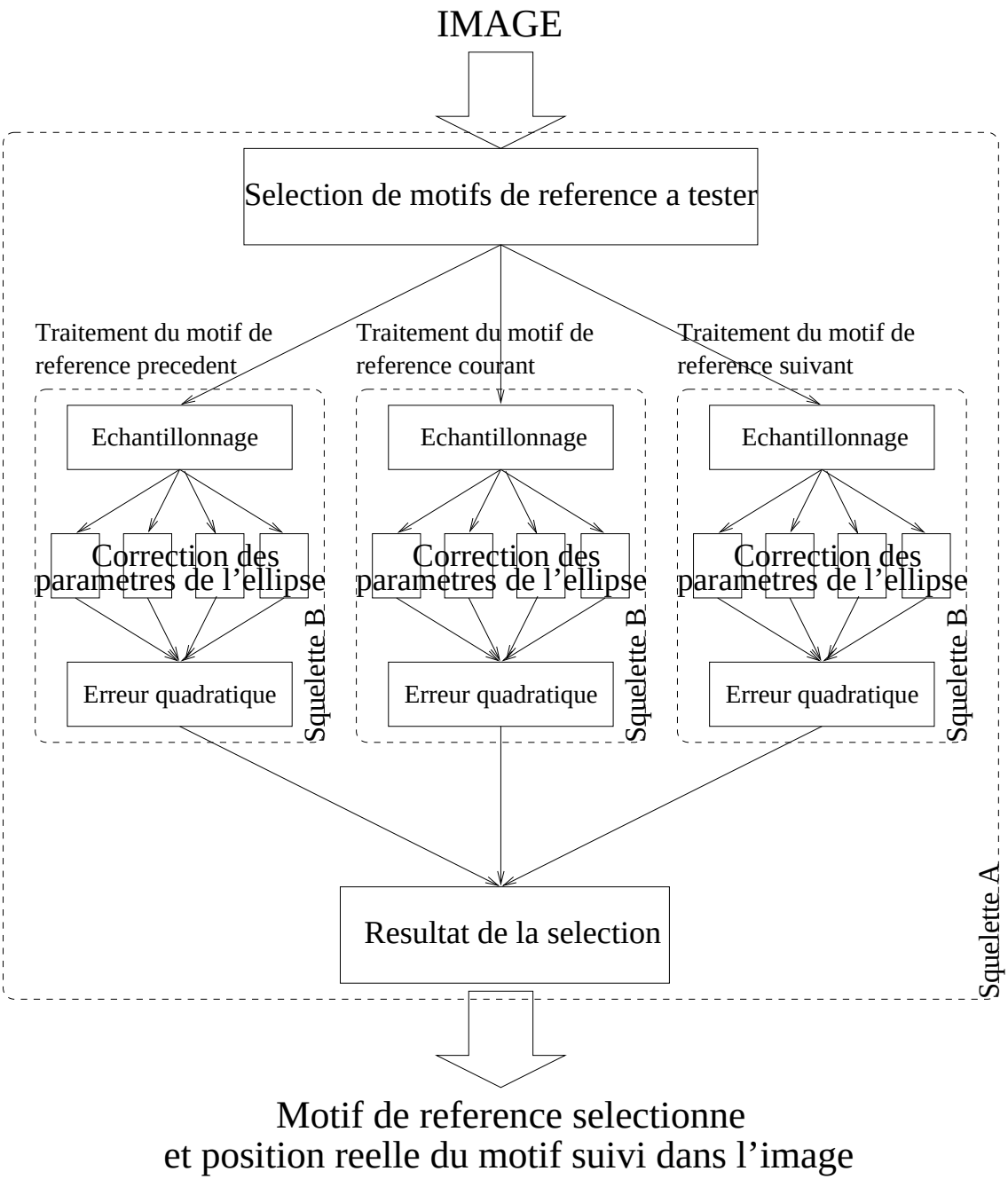


FIG. 5.30 – *Structure générale de la version parallèle de l'algorithme de suivi.*

Il est important de noter que ces deux niveaux de parallélisation ne peuvent pas être fusionnés en un seul et même niveau. Cela tient au fait que les trois multiplications matricielles (une pour le traitement de chaque motif de référence) ne peuvent elle-mêmes être fusionnées en une seule. Plus précisément, nous avons à considérer trois matrices d'interaction (matrices A des sections précédentes) et trois vecteurs de niveaux de gris (nommés $Vdiff$). Pour chaque motif de référence pris isolément, seul un vecteur doit être multiplié par une unique matrice. Fusionner les trois matrices en une seule et les trois vecteurs en un seul conduirait à effectuer des calculs inutiles, voire à des résultats erronés.

Que ce soit à l'un ou à l'autre des deux niveaux de parallélisation, les traitements sont de nature régulière, c'est-à-dire que leur complexité calculatoire ne dépend pas du contenu des données, mais seulement de leur taille, taille qui est connue lors de la compilation. Cela autorise l'emploi de squelettes SCM comme schémas de parallélisation pour l'un comme pour l'autre des deux niveaux. Par conséquent, la structure de l'application est une imbrication de deux squelettes SCM, celui qui est imbriqué jouant le rôle de la fonction de calcul du squelette englobant (fonction *compute* du SCM). La fonction de répartition *split* du squelette englobant (squelette A sur la figure 5.30) sélectionne les trois motifs de référence à tester et transmet le numéro de motif, ainsi que les paramètres de la position prédite de l'ellipse, à chaque fonction *split* du SCM imbriqué (squelette B sur la figure 5.30). Nous supposons ici que l'image à traiter est disponible dans la mémoire locale de chaque unité de calcul. La fonction *split* de ces squelettes est chargée d'échantillonner le motif dans la zone délimitée par la position prédite de l'ellipse qui lui a été transmise dans l'image courante. De surcroît elle calcule les différences de niveaux de gris entre le motif courant dans l'image et le motif de référence sélectionné. Elle transmet enfin ce vecteur, accompagné de la matrice d'interaction associée et le numéro de la ligne de la matrice servant au calcul à chaque fonction *compute* du même squelette. Toutes ces fonctions *compute* sont chargées d'estimer les corrections à apporter aux paramètres de l'ellipse. Chaque fonction se charge d'établir la correction d'un seul de ces paramètres. Les résultats obtenus sont envoyés à la fonction *merge* du squelette interne qui va procéder au calcul de la nouvelle différence de niveaux de gris grâce aux paramètres de l'ellipse une fois corrigés et à l'erreur quadratique associée. La fonction *merge* du squelette englobant peut alors recueillir les trois résultats (un par squelette imbriqué). Ils contiennent les paramètres corrigés de l'ellipse, l'erreur quadratique et le numéro du motif de référence qui est associé au calcul. La fonction peut alors finalement sélectionner le motif de référence parmi les trois initiaux qui donne la plus petite erreur quadratique et donc par suite la position correcte du motif suivi dans l'image, ainsi que le numéro du nouveau motif de référence suivi.

5.5.2.2 Implantation

Une fois la structure parallèle de l'application identifiée et fixée, SKiPPER-II peut être utilisé pour obtenir l'implantation parallèle finale de l'application sur une machine cible. Connaissant les squelettes à utiliser et la manière dont ils s'organisent, il ne reste plus au programmeur de l'application qu'à fournir les fonctions séquentielles de calcul pour les intégrer aux squelettes. Comme nous l'avons vu, aucune modification n'a à être apportée aux fonctions de calcul, mais leur interface doit faire l'objet d'une adaptation pour se mettre en conformité avec le format imposé par le noyau K/II. Puisque SKiPPER-II n'impose aucune restriction sur le prototype des fonctions utilisateur pour garantir plus de souplesse dans l'écriture des applications, seules quelques lignes de code d'interface («stub-code») viennent s'intercaler entre le noyau et les fonctions utilisateur pour faire la correspondance entre leur format attendu par le noyau et leur format donné par l'utilisateur.

Nous avons représenté sur la figure 5.31 la structure graphique de l'application et la façon dont le noyau de SKiPPER-II l'encode dans sa représentation interne. Comme nous l'avons vu, la structure de données utilisée pour la représentation interne est un simple tableau écrit en C, rapidement exploitable par le noyau. Sur cette figure nous avons respecté les notations de la section 4.3.2 (page 99).

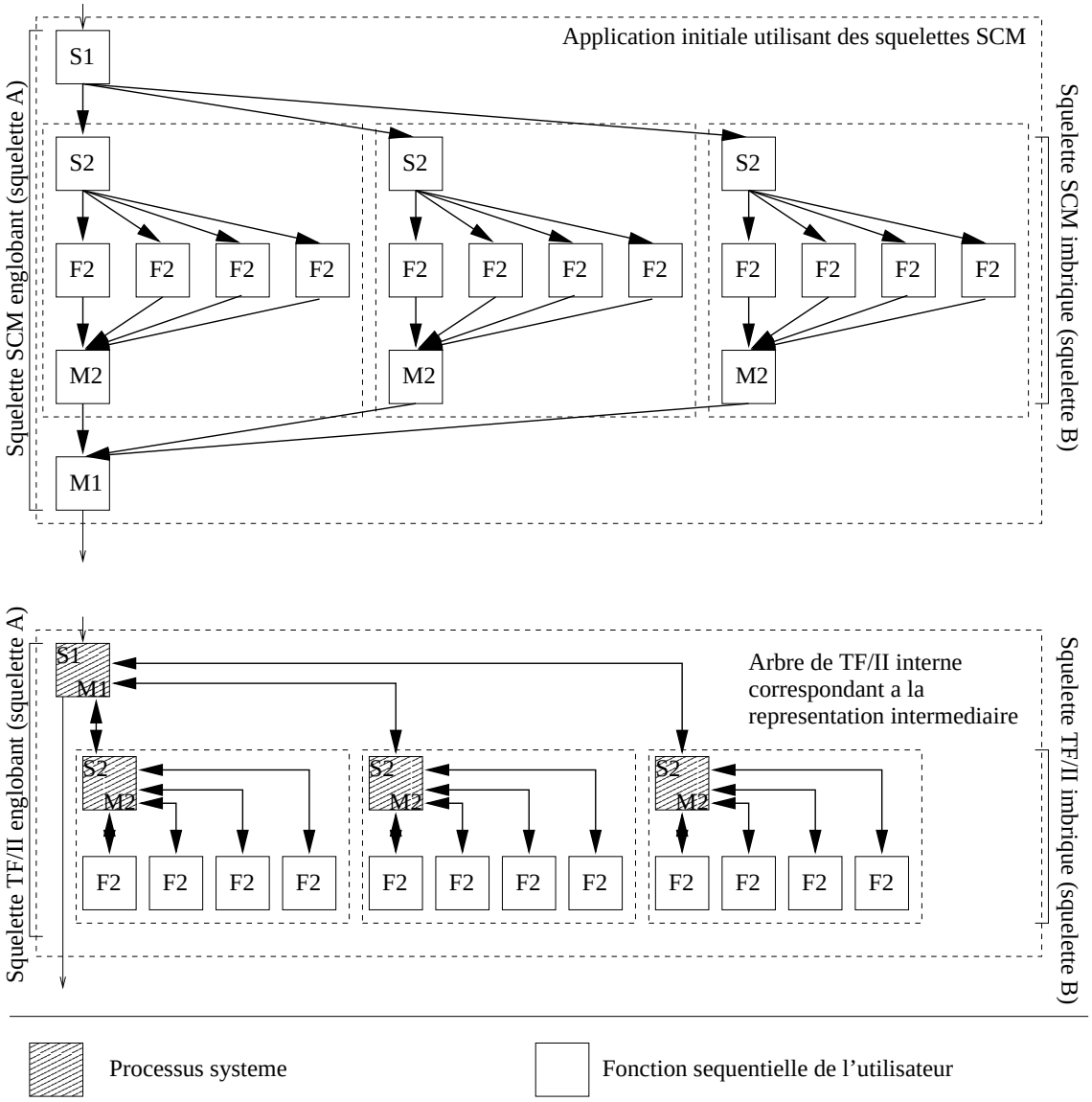
La figure 5.32 donne l'aspect des signatures des fonctions utilisateur pour cette application. On peut ici constater qu'aucune contrainte ne vient gêner leur écriture et qu'aucun ajout n'a eu besoin d'être fait par rapport à la syntaxe C normalisée.

5.5.3 Résultats de l'expérimentation

Les mesures de performances pour cette application ont été menées sur une machine Beowulf équipée de 32 nœuds de calcul de type Intel Celeron cadensés à 533 MHz. Le Beowulf était équipé d'un réseau commuté Fast Ethernet à 100 Mbits/s [9]. Les deux figures 5.33 et 5.34 montrent respectivement le temps d'exécution de l'algorithme en fonction du nombre de processeurs mis en jeu, et l'accélération relative obtenue. Deux quantités de points d'échantillonage ont été considérées pour les tests, à savoir 170 et 373 points. On pourra cependant noter qu'utiliser plus de 170 points ne donne pas de meilleurs résultats en termes de stabilité et de performances intrinsèques pour l'algorithme de suivi lui-même. Le suivi est déjà suffisamment robuste avec seulement 170 points. Le choix d'augmenter le nombre de points d'échantillonnage dans l'ellipse n'a été décidé que dans le cadre de l'évaluation des performances de SKiPPER-II (question du rapport calculs/communications qu'influence directement ici ce nombre).

Les courbes représentées sur les figures 5.33 et 5.34 montrent trois phases distinctes dans le fonctionnement du noyau de SKiPPER-II.

9. L'application a d'abord ´et´e parall´elis´ee sur un r´eseau de stations de travail Silicon Graphics de type O2 (sur lesquelles l'application avait ´et´e initialement d´evelopp´ee), puis finalement port´ee sur une machine Beowulf pour la phase d'´evaluation des performances. On notera que le haut niveau de portabilit´e de SKiPPER-II a ainsi permi d'effectuer la phase de parall´elisation dans l'environnement initial du programmeur (apr`es une simple installation de MPI), et le transfert sur une machine parall`ele seulement pour l'´evaluation des performances finale, et cela sans avoir `a modifier une seule ligne de l'application ou du noyau.



(a) Représentation graphique

```
#define SKL_NBR 2
SK2_Desc app_desc[SKL_NBR] =
{
   {SK0, END_OF_APP, MASTER, SK1 },
   {SK1, UPPER     , SLAVE , NIL }
};
```

(b) Représentation intermédiaire

FIG. 5.31 – *Représentation interne de l'application de suivi de visages*

```
S1(
  int         pattern_number,               /* Entree        */
  Ellipse     current_ellipse,              /* Entree/Sortie */
  int      ** tracker_number_to_test        /*       /Sortie */
);
S2(
  Ellipse     current_ellipse,              /* Entree/Sortie */
  int         tracker_number_to_test,       /* Entree/Sortie */
  int      *  gray_level_vector_size,       /*       /Sortie */
  float    ** gray_level_difference_vector, /*       /Sortie */
  int      ** matrix_line_number,           /*       /Sortie */
  float   *** matrix                        /*       /Sortie */
);
F2(
  Ellipse     current_ellipse,              /* Entree/Sortie */
  int         tracker_number_to_test,       /* Entree/Sortie */
  int         gray_level_vector_size,       /* Entree        */
  float    *  gray_level_difference_vector, /* Entree        */
  int         matrix_line_number,           /* Entree/Sortie */
  float    *  matrix,                       /* Entree        */
  float    *  correction                    /*       /Sortie */
);
M2(
  Ellipse     current_ellipse,              /* Entree        */
  int         tracker_number_to_test,       /* Entree/Sortie */
  int         matrix_line_number,           /* Entree        */
  float       correction                    /* Entree        */
  float    *  quadratic_error,              /*       /Sortie */
  Ellipse  *  corrected_ellipse,            /*       /Sortie */
);
M1(
  Ellipse     corrected_ellipse,            /* Entree        */
  int         tracker_number_to_test,       /* Entree        */
  float       quadratic_error,              /* Entree        */
  Ellipse  *  final_current_ellipse,        /*       /Sortie */
  int      *  final_pattern_number          /*       /Sortie */
);
```

FIG. 5.32 – *Prototypes des fonctions utilisateur pour l'application de suivi de visages*

La première de ces phases apparait lorsque seulement 2 processeurs sont utilisés (pour 373 points) et entre 2 et 3 processeurs (lorsque 170 points seulement sont utilisés). Dans cette partie des courbes on constate que le temps d'exécution est plus important lorsque plusieurs processeurs sont mis en œuvre que lorsqu'il n'y en a qu'un seul. Ce phénomène s'explique par le fait qu'avec l'utilisation de plus d'un processeur, mais néanmoins extrêmement peu, le squelette SCM englobant est déployé sur les unités de calcul. De ce fait, et conformément à ce qui a été présenté au chapitre 4, utiliser 2 nœuds de calcul au lieu d'un seul ne fournit pas plus de puissance de calcul, mais par contre accroît très sensiblement le volume des communications. Cela est dû au fait qu'un des deux nœuds de calcul est utilisé comme processus de répartition des traitements à effectuer et non pour réaliser concrètement des traitements. Lorsque le ratio calcul/communication est faible (comme dans le cas où on n'utilise que 170 points) le phénomène est très sensible et donc s'observe jusqu'à 3 nœuds. Dans ce cas, le temps passé à communiquer les données entre les processeurs annihile le gain en temps d'exécution obtenu par la mise en parallèle de ces processeurs.

De 4 à 19 nœuds, les performances augmentent avec le nombre de processeurs. Cette phase correspond au déploiement du squelette interne qui réalise la multiplication vecteur-matrice en parallèle.

La dernière phase commence quant à elle à partir de 19 nœuds. A ce moment-là, l'augmentation du nombre de processeurs ne fournit plus de puissance de calcul supplémentaire. Cela s'explique par le fait que les squelettes SCM encapsulent une stratégie de parallélisme de donnée à grain fixe, c'est-à-dire que pour se développer complètement un squelette SCM a besoin d'un nombre déterminé de processeurs fonction du travail qu'il a à effectuer mais qui ne peut être modifié pendant l'exécution. Le maximum d'efficacité est donc atteint pour un nombre de processeurs équivalent au nombre d'esclaves du SCM [10]. Passé ce nombre (ici de 19 à 32) il n'y à plus de parallélisme à exploiter, et par conséquent l'efficacité décroît d'autant. Cela implique aussi qu'en dessous du nombre de processeurs critique nécessaire à un squelette SCM, le noyau de SKiPPER-II ne peut déployer le squelette dans son intégralité. Comme nous l'avons déjà vu, il séquentialise alors une partie des esclaves et donc des calculs sur un même nœud.

Les résultats relativement faibles en terme d'efficacité qui sont relevés sur ces figures reflètent en fait un ratio calculs/communications lui aussi relativement faible dans la version parallèle. Il faut mentionner ici que la version séquentielle de l'application de suivi était déjà très efficace puisque plus rapide d'un facteur 3 que les limites que ses concepteurs s'étaient fixés en la développant. La cause principale en est la faible représentation de calculs intensifs dans l'algorithme. C'est tout particulièrement vrai pour la multiplication matricielle faite ici au sein d'un SCM pour la version parallèle : le calcul porte seulement sur la multiplication d'une matrice $(p \times N)$ par un vecteur de dimension N.

10. Dans le cas pr´esent il faut tenir compte du fait que les squelettes SCM au moment de leur ex´ecution sont sous la forme de TF/II. Par cons´equent le nombre optimal de processeurs `a utiliser pour le squelette englobant est de 4 (3 pour les calculs et 1 pour les r´epartir), et de 5 pour le squelette imbriqu´e (4+1).

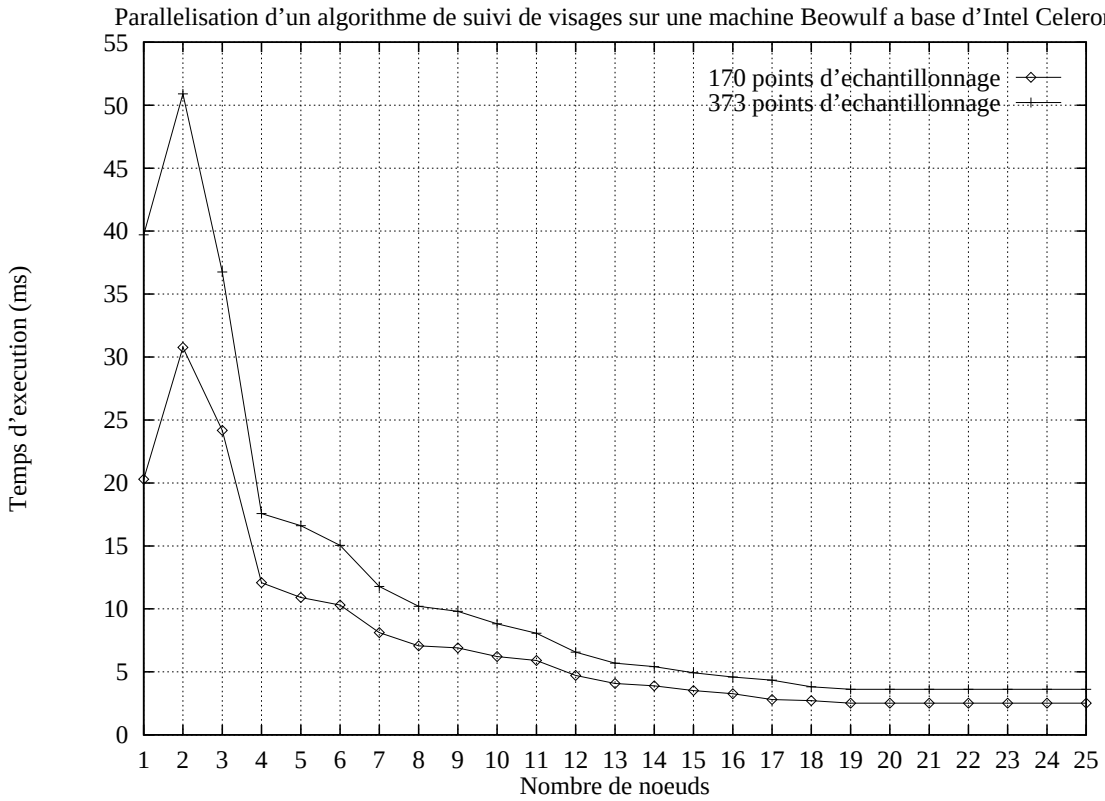


FIG. 5.33 – *Temps d'exécution en ms pour 170 et 373 points d'échantillonnage.*

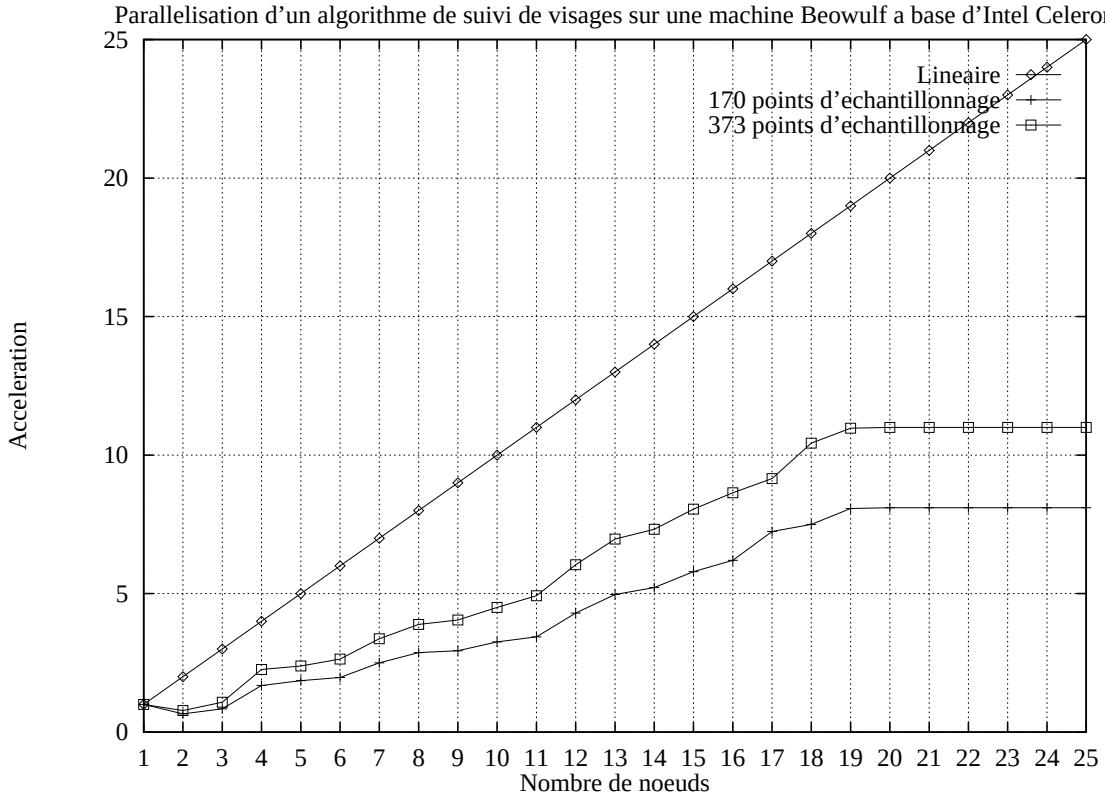


FIG. 5.34 – *Accélération pour 170 et 373 points d'échantillonnage.*

5.5.4 Discussion

La parallélisation d'une application de vision artificielle de complexité réaliste nécessitant la mise en œuvre de l'imbrication dans une méthodologie utilisant les squelettes algorithmiques a rarement été proposée à notre connaissance. Le suivi de visage que nous venons de décrire a été parallélisé en ce sens, pour montrer l'intérêt de disposer de la possibilité d'imbriquer les squelettes des bibliothèques des environnements de programmation parallèle fondés sur ce concept, afin d'élargir le champ d'application de ce type de méthodologie. Sans cette possibilité, certaines phases de l'algorithme de suivi de visages auraient dû être adaptées pour se prêter à un autre schéma de parallélisation complète, les niveaux de parallélisation que nous avons isolés ne pouvant pas être fusionnés en l'état.

Cette expérience rappelle aussi que l'imbrication de squelettes n'est pas une opération transparente en termes d'efficacité. L'utiliser par nécessité ou par commodité pour paralléliser un algorithme ne doit pas faire oublier les coûts supplémentaires que sa mise en œuvre impose.

Concernant les aspects méthodologiques de cette implantation, il faut retenir que la parallélisation complète de cet algorithme (non prévu à l'origine pour cela) n'a demandé que quelques jours de travail. Ce temps a d'ailleurs été consacré pour l'essentiel à la mise en évidence des schémas de parallélisation adéquats (sélection des squelettes et organisation de leurs interconnexions). Une fois cet important travail préparatoire terminé la deuxième étape a été d'effectuer le découpage de l'algorithme en fonctions autonomes prenant place comme fonctions de calcul dans les squelettes choisis à l'étape précédente (repartition des fonctions déjà existantes et adaptation de leurs interfaces). Par ailleurs la parallélisation de cet algorithme a confronté SKiPPER-II à la prise en charge d'un programme hétérogène au niveau de sa programmation dans le sens où certaines parties du code étaient écrites en langage C et d'autres en C++. La compilation conjointe avec SKiPPER-II n'a posé aucun problème et validé du même coup son utilisation avec le langage orienté objet qu'est le C++. De même, l'algorithme de suivi de visages faisait appel à de nombreux rendus graphiques pour la présentation des résultats, affichages programmés tout à la fois en X-Window avec les bibliothèques XtIntrinsics qu'en OpenGL. Là encore, l'utilisation de ces bibliothèques s'est faite de manière totalement transparente pour SKiPPER-II. Enfin, il est intéressant de noter que l'application faisait appel à de nombreuses fonctions développées dans des librairies extérieures (c'est-à-dire sur lesquelles le programmeur de l'application a peu d'emprise).

Concernant les choix de parallélisation pour l'algorithme de suivi que nous venons de présenter, nous pensons à la lumière du travail qui a été mené que, pour des aspects d'efficacité pure, il serait plus judicieux de paralléliser la première phase de l'algorithme, à savoir sa phase d'apprentissage «hors-ligne». La raison en est que diminuer son temps d'exécution (même pour une phase «hors-ligne») offrirait la possibilité d'exploiter les deux phases de l'algorithme «en-ligne». Les débouchés applicatifs pourraient être une utilisation dans le cadre de suivis multi-cibles temps-réel (comme par exemple pour un suivi de véhicules automobiles [MCMD98]). Dans ce cas l'accélération attendue pourrait permettre d'apprendre les motifs de référence «à la volée» et ainsi éviter l'utilisation d'une base de données extrêmement volumineuse lorsqu'il est nécessaire à l'algorithme de s'adapter à un environnement changeant.

Conclusion

Le travail que nous avons présenté dans ce mémoire poursuit le développement du projet SKiPPER (SKeleton-based Parallel Programming EnviRonment) initié en 1997.

SKiPPER est un environnement d'aide à la programmation parallèle, conçu en vue du prototypage rapide d'applications de vision artificielle sur des machines parallèles de type MIMD-DM. La méthodologie de programmation retenue est fondée sur l'utilisation des squelettes algorithmiques. SKiPPER propose au programmeur un jeu de squelettes répondant aux problèmes rencontrés en vision artificielle, sur la base d'une étude d'applications préalablement parallélisées «à la main». Le jeu de squelettes permet au programmeur de décrire l'organisation de son application en terme de parallélisme, l'objectif final étant de réduire le temps de conception, implantation et validation d'algorithmes sur une machine parallèle.

A travers l'outil SKiPPER, cette méthode de programmation veut donc faciliter la programmation parallèle en prenant en charge ses aspects bas-niveau (découpage de l'application en processus, placement et ordonnancement de ces processus, routage des communications,...)

L'achèvement du développement de la version SKiPPER-I de l'outil en 1999, avait permi de proposer un environnement répondant aux exigences du prototypage rapide.

Avec SKiPPER-I, la composition de squelettes, *i.e.* la possibilité de combiner plusieurs squelettes pour paralléliser une application, était limitée au séquencement. L'imbrication, sauf cas particulier, n'était pas possible. Cette possibilité d'imbrication est toutefois apparue souhaitable dans l'optique de la parallélisation d'applications de vision complexes.

Cette constatation a motivé le développement d'une nouvelle version de l'environnement, SKiPPER-II, qui fait l'objet de ce mémoire. Nous avons donc proposé un nouveau modèle d'exécution pour les squelettes de SKiPPER. Ce modèle est bâti autour d'un «méta-squelette» interne (TF/II) pouvant exprimer le comportement de n'importe lequel des squelettes proposés au programmeur dans SKiPPER. Son rôle premier est de simplifier l'implantation de la notion d'imbrication de squelettes. En effet, on réduit ainsi les combinaisons potentielles d'imbrication entre squelettes. Qui plus est, il est conçu explicitement pour pouvoir être imbriqué, et offre donc des possibilités de composition systématique des squelettes.

Le modèle d'exécution de ce nouveau squelette étant intrinsèquement dynamique, le nouveau noyau de SKiPPER architecturé pour l'exécuter l'est aussi. Les difficultés de prise en charge des squelettes dynamiques de notre jeu de squelettes dans la version précédente sont résolues avec SKiPPER-II naturellement par l'emploi d'un modèle d'exécution complètement dynamique. Le revers d'un tel modèle est la difficulté de prédire ses performances, de nombreux paramètres entrant en jeu et, de fait pour certains, non dépendants de l'application seule ou de l'architecture seule, mais de leur interaction au moment de l'exécution.

Par ailleurs, le «méta-squelette» TF/II fourni un moyen de distinguer la base de squelettes mise à la disposition du programmeur de la couche exécutable de l'environnement chargée

de leur implantation effective. Cette nouvelle capacité de l'environnement offre la possibilité de proposer d'autres jeux de squelettes, adaptés éventuellement à d'autres champs applicatifs, sans nécessité aucune modification du noyau de SKiPPER-II.

Enfin, maintenant que la partie opérationnelle de SKiPPER-II est devenue indépendante du jeu de squelettes utilisé, cette version fournit également une approche totalement portable de l'environnement ne nécessitant, grâce à l'utilisation de MPI, aucune modification pour être installée sur des plate-formes parallèles différentes.

La question peut se poser de savoir s'il reste souhaitable de continuer à proposer au programmeur un jeu de squelettes spécifique, variable selon son domaine applicatif, ou s'il est préférable de le remplacer par le seul TF/II. Si les deux approches ont leurs avantages et leurs inconvénients, il nous apparaît que la première est intéressante méthodologiquement. En effet, chaque squelette exprime un schéma particulier de parallélisme, et donc donne une représentation particulière de l'organisation de l'application. Ainsi, de notre point de vue, le programmeur peut plus facilement appréhender la vision parallèle de son application que si on lui proposait un squelette «à tout faire» dans lequel le sens de tel ou tel schéma logique de parallélisation serait noyé. Conserver un jeu de squelettes pour le programmeur, et ne pas le remplacer par un squelette générique, c'est conserver la capacité qu'il a à trouver un sens à sa parallélisation et donc la faciliter.

SKiPPER est devenu une chaîne complète de développement en programmation parallèle, s'accommodant de plates-formes de plus en plus nombreuses, et affichant un potentiel à être utilisé dans d'autres domaines applicatifs que la vision artificielle.

Concernant la version SKiPPER-II que nous avons présentée, les développements futurs devront mettre l'accent sur la partie frontale, mais pas uniquement pour la doter d'une version complètement automatisée de la traduction des squelettes en TF/II. En effet, le développement de cette partie doit être orienté vers la prise en compte de jeux multiples de squelettes, dédiés à des domaines applicatifs variés, voire en ayant à l'esprit la possibilité d'offrir au programmeur la possibilité de spécifier lui-même de nouveaux squelettes, chose rendue possible par une partie opérationnelle de SKiPPER-II (le noyau, exécutant les squelettes) totalement indépendante des squelettes eux-mêmes. Cette dernière capacité serait destinée à deux objectifs. Le premier serait, pour le spécialiste d'un domaine applicatif, d'évaluer le bien fondé de nouveaux squelettes pour son domaine, puis de proposer à des experts un prototype des squelettes dont il a besoin, afin de créer une base de squelettes efficace et commune à sa spécialité (et ainsi y diffuser l'utilisation de la programmation parallèle). La seconde serait d'offrir au programmeur «occasionnel», c'est-à-dire ne s'investissant dans un domaine applicatif que ponctuellement, la possibilité de définir un squelette adapté à son besoin à un instant donné, besoin qui ne serait pas satisfait par les bases de squelettes dont il dispose, sans être tributaire de l'émergence d'un nouveau jeu de squelettes le prenant en compte, dans la mesure où le besoin est ponctuel.

Concernant la maturité du projet dans sa globalité, il apparaît maintenant intéressant d'envisager une étude d'envergure sur l'apport qu'il peut fournir à une communauté d'utilisateurs, en le comparant à une approche «manuelle» ainsi qu'à d'autres outils équivalents. Cette étude n'aurait plus pour objet des questions de performances temporelles, de capacités de gestion des architectures matérielles ou de l'efficacité de telle ou telle technique d'implantation, mais d'ergonomie, de méthode et d'efficacité de travail et d'utilisation. Ses prétentions étant d'aider à la parallélisation d'applications, il s'agit maintenant d'ouvrir son utilisation en dehors de la communauté restreinte de ses développeurs et des spécialistes travaillant avec eux, pour tirer des enseignements sur la manière dont un tel environnement est perçu et utilisé, non plus au

coup par coup, mais régulièrement. Cela peut-être envisagé notamment dans notre milieu universitaire par sa mise à disposition dans l'enseignement de la programmation parallèle auprès des étudiants. Couplée avec un enseignement «classique» sur la programmation parallèle, cette approche peut permettre de dégager les apports et les limites d'une telle approche auprès d'un panel large d'usagers novices. Une telle étude permettrait de fournir des données statistiques et comportementales sur l'impact des méthodes de programmation parallèle défendues par le projet SKiPPER dans une communauté d'utilisateurs appelée à les utiliser de manière systématique par la suite, et par la-même de les faire évoluer.

Bibliographie

[AH87] S. Abramsky et C. Hankin. Abstract Interpretation of Declarative Languages. *Ellis Horwood, Chichester, West Sussex*, 1987.

[Ale77] Ch. Alexander. *A Pattern Language*. Oxford University Press, 1977.

[Ale79] Ch. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[AMC+00] R. Aufrere, F. Marmoiton, R. Chapuis, F. Collange, et J.-P. Dérutin. Détection de route et suivi de véhicules par vision pour l'ACC. *Traitement du Signal*, 17(3), 2000.

[AOC+88] G. Andrews, R.A. Olsson, M.A. Coffin, I. Elshoff, K. Nilsen, T. Purdin, et G. Townsend. An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, 1988.

[BCD+97] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, et M. Vanneschi. An Environment for Structured Parallel Programming. *Advances in High Performance Computing*, pages 219–234, 1997.

[BCP+98] B. Bacci, B. Cantalupo, P. Pesciullesi, R. Ravazzolo, A. Riaudo, et L. Vanneschi. SKIE: User's Guide (version 2.0). Rapport technique, QSW Ltd. Rome, Italie, Décembre 1998.

[BDHR94] A. Bellon, J.-P. Dérutin, F. Heitz, et Y. Ricquebourg. Real-time Collision Avoidance at Road-crossings on Board of the Prometheus Prolab-2 Vehicle. In *Intelligent Vehicles Symposium*, Paris, Octobre 1994.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, et M. Vanneschi. P^3L: A Structured High-level Programming Language and Its Structured Support. *Concurrency Practice and Experience*, 7(3):225–255, Mai 1995.

[BDPV99] B. Bacci, M. Danelutto, S. Pelagatti, et M. Vanneschi. SKIE: A Heterogeneous environment for HPC Applications. *Parallel Computing*, 1999.

[BE01] A. Beuraud et Y. Elguetioui. Intégration du standard IDL dans l'environnement de programmation parallèle SKiPPER-II. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Mars 2001. Rapport de projet de deuxième année d'élève-ingénieur ISIMA.

[BHS+93] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, et S. Chaterjee. Implementation of Portable Nested Data-parallel Language. In *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, Californie, USA, Octobre 1993. ACM Press.

[BKT92] H. Bal, M. Kaashoek, et A. Tannenbaum. Orca: a Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[BM01] D. Becker et P. Merkey. The Beowulf Project. http://www.beowulf.org, Avril 2001.

[BN00] A. Béthune et N. Naudier. Développement d'un serveur de requêtes gérant le bus PCI pour la machine parallèle TransAlpha. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Mars 2000. Rapport de projet de deuxième année d'élève-ingénieur ISIMA.

[Boa01] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface. Rapport technique, OpenMP Architecture Review Board, Novembre 2001. Version 2.0 (Draft 11.05).

[Bra95] T. Bratvold. *Skeleton-based parallelisation of functional programs*. Thèse d'université, Departement de Génie Informatique et Electronique, Université d'Edimbourg, Ecosse, Juillet 1995.

[Can93] R. Canals. *Implantation d'algorithmes de segmentation d'images sur la machine parallèle Transvision*. Thèse d'université, Université Blaise-Pascal, Clermont-Ferrand II, Octobre 1993.

[CDS+] R. Coudarcher, F. Duculty, J. Sérot, F. Jurie, J.-P. Dérutin, et M. Dhome. Parallelisation of a Face Tracking Algorithm with the SKiPPER-II Parallel Programming Environment. In *Soumis à IEEE CAMP 2002 International Workshop on Computer Architectures for Machine Perception, Indian Institute of Technology Bombay, Mumbai, Inde*.

[Cha91] F. Chantemargue. *Segmentation d'images par approche de type division-fusion*. Thèse d'université, Université Blaise-Pascal, Clermont-Ferrand II, Décembre 1991.

[Che93] D.Y. Cheng. A Survey of Parallel Programming Languages and Tools. Rapport technique RND-93-005, NASA Ames Research Center, Californie, USA, Mars 1993.

[CMT94] L. Colombet, Ph. Michallon, et D. Trystram. Parallel Matrix-Vector Product on Rings with a Minimum of Communications. Rapport technique RR 10, IMAG, Projet APACHE, Grenoble, Juin 1994.

[Col89] M. Cole. *Algorithmic skeletons: structured management of parallel computations*. Pitman/MIT Press, 1989.

[Col99] M. Cole. Algorithmic Skeletons (Chapitre 13). In *Research Directions in Parallel Functional Programming*. G.J. Michaelson and K. Hammond. Springer-Verlag, 1999.

[Cou96] R. Coudarcher. Portage d'une chaîne de segmentation au sens contour sur machine de vision artificielle multi-T9000. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Septembre 1996. Rapport de stage de deuxième année d'élève-ingénieur ISIMA.

[Cou97] R. Coudarcher. Parallélisation de la phase d'extraction de groupements perceptifs dans une chaîne de segmentation d'images pour machine MIMD à mémoire distribuée. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Septembre 1997. Rapport de stage de DEA «Electronique et Systèmes» et de troisième année d'élève-ingénieur ISIMA.

[CR95] M.C. Carlisle et A. Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38. ACM Press, Juillet 1995.

[CSD00] R. Coudarcher, J. Sérot, et J.-P. Dérutin. Mise en œuvre statique de squelettes de parallélisation dynamiques sous SynDEx. In *$5^{ème}$ Workshop AAA sur l'Adéquation Algorithme Architecture*, pages 189–196, INRIA de Rocquencourt, Janvier 2000. INRIA.

[Dan99] M. Danelutto. Dynamic Run Time Support for Skeletons. *Proceedings of the ParCo99 Conference, Delft, Pays-Bas*, Août 1999.

[Dan01] M. Danelutto. On skeletons and design patterns. In *PARCO 2001. Special Session on Parallel and Distributed Image and Video Processing (ParIm'o1)*, Naples, Italie, Septembre 2001.

[DCDH90] J. Dongarra, J. Du Croz, I. Duff, et S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[DDJ01] F. Duculty, M. Dhome, et F. Jurie. Tracking of 3D Objects From Appearance. In *SCIA 2001*, volume 1, pages 515–522, 2001.

[DGT93] J. Darlington, M. Ghanem, et H.W. To. Structured Parallel Programming. In *Proceedings of MPPM*, Berlin, 1993.

[DGTY95a] J. Darlington, Y. Guo, H.W. To, et J. Yang. Functional Skeletons for Parallel Coordination. In *Proceedings of EuroPar'95*, pages 55–69. Springer-Verlag, Août 1995.

[DGTY95b] J. Darlington, Y. Guo, H.W. To, et J. Yang. Parallel Skeletons for Structured Composition. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–28. ACM Press, 1995.

[DOPV94] M. Danelutto, S. Orlando, S. Pelagatti, et M. Vanneschi. Parallel Programming Models Based on the Restricted Computation Structure Approach. Rapport technique R/3/133, Université de Pise, Pise, Italie, 1994.

[DT93] J. Darlington et H.W. To. Building Parallel Applications without Programming. *Abstract Machine Models*, 1993.

[Fly66] M.J. Flynn. Very High Speed Computers. *Proceedings of IEEE*, 54, Décembre 1966.

[Fly96] M.J. Flynn. Parallel Processors Were the Future... and May Yet Be. *IEEE Computer*, pages 151–152, Decembre 1996.

[For94] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3-4), 1994.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, et V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[GG91] G. Giraudon et P. Garnesson. Polygonal Approximation Overview and Perspectives. Rapport technique 1621, INRIA programme 4 : robotique, image et vision, INRIA Sophia-Antipolis, Juin 1991.

[GHJV94] E. Gamma, R. Helm, R. Johnson, et J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gin95] D. Ginhac. Spécification et implantation d'un algorithme flot de données d'étiquetage en composantes connexes sur la machine multiprocesseurs à mémoire distribuée Transvision. Rapport technique, Université Blaise-Pascal, Clermont-Ferrand, Juin 1995. Mémoire de DEA.

[Gin99] D. Ginhac. *Prototypage rapide d'applications parallèles de vision artificielle par squelettes fonctionnels*. Thèse d'université, Université Blaise-Pascal, Clermont-Ferrand II, Janvier 1999.

[GLS94] W. Gropp, E. Lusk, et A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing*. MIT Press, 1994.

[GLS98] T. Grandpierre, C. Lavarenne, et Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Rapport technique, INRIA, Rocquencourt, Août 1998. Rapport de recherche RR 3476.

[GLS99] T. Grandpierre, C. Lavarenne, et Y. Sorel. Optimized Rapid Prototyping For Real Time Embedded Heterogeneous Multiprocessors. *CODES'99 7th International Workshop on Hardware/Software Co-Design, Rome*, Mai 1999.

[GMN+94] A. Goldberg, P. Mills, L. Nyland, J. Prins, J. Reif, et J. Riely. Specification and Development of Parallel Algorithms with the Proteus System. *ACM Press*, DIMACS: Specification of Parallel Algorithms, 1994.

[GMW79] M.J. Gordon, R. Milner, et C. Wadsworth. Edinburgh LCF. LNCS 78, 1979.

[GSD98] D. Ginhac, J. Sérot, et J.-P. Dérutin. Fast Prototyping of Image Processing applications Using Functional Skeletons on MIMD-DM Architecture. In *IAPR Workshop on Machine Vision Applications*, pages 468–471, Novembre 1998.

[GSDC99] D. Ginhac, J. Sérot, J.-P. Dérutin, et R. Chapuis. SKiPPER : un environnement de programmation parallèle fondé sur les squelettes et dédiés au traitement d'images. In *17eme Colloque GRETSI sur le traitement du signal et des images*, 13-17 Septembre 1999.

[GSP98] D. Goswami, A. Singh, et B.R. Preiss. A Skeleton-based Model for Developing Parallel Applications Using a Network of Processors. http://www.pads.uwaterloo.ca/Bruno.Preiss/page69.html, Janvier 1998.

[GSP99] D. Goswami, A. Singh, et B.R. Preiss. Architectural Skeletons: The Re-usable Building-Blocks for Parallel Applications. In *Proceedings 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1250–1256, Las Vegas, Nevada, USA, Juin 1999. Computer Science Research, Education and Applications Technology, volume 3.

[GSP00] D. Goswami, A. Singh, et B.R. Preiss. Building Parallel Applications Using Design Patterns. In *Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation*, New York, USA, Juillet 2000. Springer-Verlag.

[GSP01] D. Goswami, A. Singh, et B.R. Preiss. From Design Patterns to Parallel Architectural Skeletons. *Journal of Parallel and Distributed Computing*, 61(6), Juin 2001.

[GT00] Q. Guzel et J. Tiré. Implantation d'une interface MPI sur une machine parallèle MIMD-DM dédiée à la vision artificielle temps réel. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Mars 2000. Rapport de projet de deuxième année d'élève-ingénieur ISIMA.

[Ham00] M.M. Hamdan. *A Combinational Framework for Parallel Programming using Algorithmic Skeletons*. Thèse d'université, Departement Génie Informatique et Electrique, Université Heriot-Watt, Edimbourg, Ecosse, Janvier 2000.

[HML96] J. Hoffman et J. Margerum-Leys. Rapid Prototyping as an Instructional Design. *http://www.umich.edu/˜jmargeru/prototyping*, 1996. Collection d'articles.

[HP74] S. Horowitz et T. Pavlidis. Picture Segmentation by a Direct Split and Merge Procedure. In *Second International Conference on Pattern Recognition*, pages 424–433, 1974.

[HPF99a] P. Hudak, J. Peterson, et J.H. Fasel. A Gentle Introduction to Haskell 98. http://www.haskell.org/tutorial, Octobre 1999.

[HPF99b] P. Hudak, J. Peterson, et J.H. Fasel. A Gentle Introduction to Haskell 98. Rapport technique, Universités de Yale et de Californie, Octobre 1999.

[INR02] INRIA. Le langage Caml. http://caml.inria.fr/index-fra.html, Août 2002.

[JD01] F. Jurie et M. Dhome. A Simple and Efficient Template Matching Algorithm. In *ICCV 2001*, volume 2, pages 544–549, 2001.

[Keb99] C. Kebler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. In *Proc. of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 613–619, Las Vegas, USA, Juillet 1999. CSREA Press.

[Kep01] J. Kepner. Exploiting VSIPL and OpenMP for Parallel Image Processing. In *Astronomical Data Analysis Software and Systems X*, ASP San Francisco, 2001. in ASP Conf. Ser., Vol. 238.

[KHP+95] K.M. Kavi, A.R. Hurson, P. Patadia, E. Abraham, et P. Shanmugam. Design of Cache Memories for Multi-threaded Dataflow Architecture. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 253–264, 1995.

[KS97] C. Kebler et H. Seidel. The Fork95 Parallel Programming Language: Design, Implementation, Application. *International Journal of Parallel Programming*, 25(1):17–49, Fevrier 1997.

[LCD94a] P. Legrand, R. Canals, et J.-P. Dérutin. Edge and Region Segmentation Processes on the Parallel Vision Machine Transvision. In *Computer Architecture for Machine Perception (CAMP 93)*, pages 410–420, Nouvelle-Orléans, USA, Décembre 1994.

[LCD94b] P. Legrand, R. Canals, et J.-P. Dérutin. Méthodes SPMD d'implantation de chaînes de traitement d'images sur la machine Transvision. In *$9^{ème}$ Congrès de Reconnaissance des Formes et d'Intelligence Artificielle (RFIA 94)*, Paris, Janvier 1994.

[Leg95] P. Legrand. *Schémas de parallélisation d'applications de traitement d'images sur la machine parallèle Transvision*. Thèse d'université, Université Blaise-Pascal, Clermont-Ferrand II, Décembre 1995.

[Ler01] E.J. Lerner. Cellular Architecture Builds Next Generation Supercomputers. *IBM Think Research*, 2001. Journal électronique d'IBM Research.

[LSS91] C. Lavarenne, O. Seghrouchi, et Y. Sorel. The SynDEx Software Environment for Real-time Distributed Systems Design and Implementation. In *Proc. of the European Control Conference*, Juillet 1991.

[Mac91] D. MacKenzie. The Influence of the Los Alamos and Livermore National Laboratories on the Development of Supercomputing. *Proceedings of IEEE Annals of the History of Computing*, 13(2):179–201, Avril-Juin 1991.

[Mar00] F. Marmoiton. *Détection et suivi par vision monoculaire d'obstacles mobiles coopératifs à partir d'un véhicule expérimental automobile*. Thèse d'université, Université Blaise-Pascal, Clermont-Ferrand II, Janvier 2000.

[MCMD98] F. Marmoiton, F. Collange, Ph. Martinet, et J.-P. Dérutin. A Real Time Car Tracker. In *International Conference on Advances in Vehicle Control and Safety*, Juillet 1998.

[Mil84] R. Milner. A Proposal for Standard ML. In *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pages 184–197, Août 1984.

[MMHB96] N. MacDonald, E. Minty, T. Harding, et S. Brown. *Writing Message-Passing Parallel Programs with MPI. A two-day course*. EPCC, Université d'Edimbourg, Ecosse, 1996.

[MMS99a] B.L. Massingill, T.G. Mattson, et B.A. Sanders. A Pattern Language for Parallel Application Programming. Rapport technique, Université de Floride, 1999. UF CISE TF 99-022.

[MMS99b] B.L. Massingill, T.G. Mattson, et B.A. Sanders. Patterns for Parallel Application Programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, 1999.

[MMS00] B.L. Massingill, T.G. Mattson, et B.A. Sanders. A Pattern Language for Parallel Application Programs. In W. Karl A. Bode, T. Ludwig et R. Wissmuller, éditeurs, *Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 678–681. Springer-Verlag, 2000.

[MNP+91] P. Mills, L. Nyland, J. Prins, J. Reif, et R. Wagner. Prototyping Parallel and Distributed Programs in Proteus. In *Proceedings of Third IEEE Symposium on Parallel and Distributed Processing*, 1991.

[MS95] G.J. Michaelson et N.R. Scaife. Prototyping a parallel vision system in Standard ML. *Journal of Functional Programming*, 5(3):345–382, Juillet 1995.

[MSBK00] G. Michaelson, N. Scaife, P. Bristow, et P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications, Special issue on High Level Models and Languages for Parallel Processing*, 2000.

[MSSB00] S. McDonald, D. Szafron, J. Schaeffer, et S. Bromling. Generating Parallel Program Frameworks from Parallel Design Patterns. In W. Karl A. Bode, T. Ludwig et R. Wissmuller, éditeurs, *Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 95–105. Springer-Verlag, 2000.

[MTH90] R. Milner, M. Tofte, et R. Harper. The Definition of Standard ML. MIT Press, 1990.

[NP92] L.S. Nyland et J.F. Prins. Prototyping Parallel Algorithms. In Panagiotis Metaxas Donald Johnson, Fillia Makedon, éditeur, *Proc. of the 1992 DAGS/PC Symposium*, pages 31–39, Dartmouth College, Hanovre, NH, Juin 1992. Dartmouth College.

[NP00] D.S. Nikolopoulos et T.S. Papatheodorou. Fast Synchronization on Scalable Cache-Coherent Multiprocessors using Hybrid Primitives. In *IEEE/ACM 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexique, Mai 2000.

[NPP98] D.S. Nikolopoulos, E.D. Polychronopoulos, et T.S. Papatheodorou. Enhancing the Performance of Autoscheduling with Locality-based Partitioning in Distributed Shared Memory Multiprocessors. In *Proceedings of the 4th EuroPar Conference*, volume LNCS 1470, pages 491–501, Southampton, Grande-Bretagne, Septembre 1998.

[Nyl91] L.S. Nyland. *The Design of a Prototyping Programming Language for Parallel and Sequential Algorithms*. Thèse d'université, Duke University, USA, 1991.

[PAR96a] PARSYS. The PARSYS TransAlpha Cross Development Toolset - Reference Manual. Rapport technique, PARSYS Ltd., Grande-Bretagne, Avril 1996. Doc. v2.0 (réf. 2180-0100).

[PAR96b] PARSYS. The PARSYS TransAlpha Module TA9108 - Data Sheet. Rapport technique, PARSYS Ltd., Grande-Bretagne, Septembre 1996. Doc. v1.4 (réf. 2180-0300).

[Pau91] L.C. Paulson. *ML for the Working Programmer*. CUP, 1991.

[Pel93] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs*. Thèse d'université, Departement d'informatique, Université de Pise, Italie, Mars 1993.

[Pel97] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, Londre, 1997.

[PH94] D. Patterson et J. Hennessy. *Organisation et conception des ordinateurs. L'interface matériel/logiciel*, pages 604–660. Dunod, 1994.

[Ran95] R. Rangaswami. HOPP - A Higher Order Parallel Programming Model. In *Marc Moonen (ed), Algorithms and Parallel VLSI Architectures. Elsevier*, 1995.

[Ran96] R. Rangaswami. *A Cost Analysis for a Higher-Order Parallel Programming Model*. Thèse d'université, Departement de Génie Informatique et Electronique, Université d'Edimbourg, Ecosse, 1996.

[Roy96] P. Van Roy. La disparition de l'ordinateur. Rapport technique, Université catholique de Louvain, Département d'Ingénierie Informatique, Belgique, Octobre 1996. Exposé donné au Conseil de la Faculté.

[RR96] T. Rauber et G. Rünger. The Compiler TwoL for the Design of Parallel Implementations. In *Proc. 4th conference on parallel architectures and compilation techniques*, pages 282–301, Boston, USA, Octobre 1996. IEEE Computer Society Press.

[RS01] B.R. Rau et M.S. Schlansker. Embedded Computer Architecture and Automation. *Computer*, 34(4):75–83, Avril 2001.

[SBMK98] N. Scaife, P. Bristow, G. Michaelson, et P. King. Engineering a Parallel Compiler for SML. *Clack, T. Davie and K. Hammond (eds): Proceedings of 10th International Workshop on Implementation of Functional Languages, University College, Londre, UK*, pages 213–226, Septembre 1998.

[SE94] S. Subramaniam et D. Eager. Affinity Scheduling of Unbalanced Workloads. In *Supercomputing'94*. IEEE Computer Society, 1994.

[Sér01] J. Sérot. CamlFlow: a Caml to Data-Flow Translator. In *Trends in Functional Programming*, volume 2. S. Gilmore. INTELLECT, 2001.

[Sér02] J. Sérot. *Prototypage rapide d'applications de vision artificielle sur architectures parallèles dédiées*. Thèse d'université, (Habilitation à diriger des recherches), LASMEA, Université Blaise-Pascal, Clermont-Ferrand II, Avril 2002.

[SGCD01] J. Sérot, D. Ginhac, R. Chapuis, et J.P. Dérutin. Fast Prototyping of Parallel Vision Applications Using Functional Skeletons. *Journal of Machine Vision and Applications*, 12(6):271–290, Juin 2001.

[SGD99] J. Sérot, D. Ginhac, et J.-P. Dérutin. SKiPPER: A SKeleton-based Parallel Programming EnviRonment for Real-time Image Processing Applications. In V. Malyshkin, éditeur, *5th International Conference on Parallel Computing Technologies (PaCT-99)*, volume 1662 of *LNCS*, pages 296–305. Springer, Septembre 1999.

[SK00a] F.J. Seinstra et D. Koelma. Accurate Performance Models of Parallel Low Level Image Processing Operations Based on a Simple Abstract Machine. Rapport technique, Intelligent Sensory Information Systems, Departement d'informatique, Université d'Amsterdam, Pays-bas, Septembre 2000. ISIS Internal Report.

[SK00b] F.J. Seinstra et D. Koelma. Transparent Parallel Image Processing by way of a Familiar Sequential API. In *Proceedings of the International Conference on Pattern Recognition (ICPR'00)*. IEEE, 2000.

[SKG00] F.J. Seinstra, D. Koelma, et J.M. Geusebroek. A Software Architecture for User Transparent Parallel Image Processing. Rapport technique, Intelligent Sensory Information Systems, Departement d'informatique, Université d'Amsterdam, Pays-bas, Décembre 2000. ISIS Technical Report Series, Vol. 14.

[Ski90] D.B. Skillicorn. Architecture-independent Parallel Computation. *IEEE Computer*, 23(12):38–50, Décembre 1990.

[Slo82] D.L. Slotnick. The Conception and Development of Parallel Processors - A Personal Memoir. *IEEE Annals of the History of Computing*, 4(1):20–30, Janvier 1982.

[SMH01] N. Scaife, G. Michaelson, et S. Horiguchi. Comparative Cross-Platform Performance Results from a Parallelizing SML Compiler. In T. Arts et M. Mohnen, éditeurs, *Proceedings of 13th International Workshop on the Implementation of Functional Languages*, volume LNCS 2312, pages 138–154, Stockholm, Septembre 2001. Springer-Verlag.

[SMW96] N.R. Scaife, G.J. Michaelson, et A.M. Wallace. A Method for Developing Parallel Vision Algorithms with an Example of Edge Tracking. Rapport technique, Département d'ingénierie informatique et electronique, Université Heriot-Watt, Edimbourg, Ecosse, Janvier 1996. Rapport de recherche RM/96/1.

[Sor94] Y. Sorel. Massively Parallel Systems with Real Time Constraints. The "Algorithm Architecture Adequation" Methodology. In *Proc. Massively Parallel Computing Systems*, Ischia, Italie, Mai 1994.

[Sor96] Y. Sorel. Real-time Embedded Image Processing Application Using the AAA Methodology. In *IEEE International Conference on Image Processing*, Novembre 1996.

[Sor02] Y. Sorel. Méthodologie Adéquation Algorithme Architecture - AAA et SynDEx. http://www-rocq.inria.fr/syndex, Avril 2002.

[SS94] D. Szafron et J. Schaeffer. Experimentally Assessing the Usability of Parallel Programming Systems. *Programming Environments for Massively Parallel Distributed Systems*, pages 195–201, 1994. Birkhauser-Verlag.

[SS96] D. Szafron et J. Schaeffer. An Experiment to Measure the Usability of Parallel Programming Systems. *Concurrency: Practice and Experience*, 8(2):146–166, 1996.

[SSLP93] J. Schaeffer, D. Szafron, G. Lobe, et I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, Août 1993.

[SSS98] A. Singh, J. Schaeffer, et D. Szafron. Experience with Parallel Programming Using Code Templates. *Concurrency Practice and Experience*, 10(2):91–120, Février 1998.

[ST98] D.B. Skillicorn et D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, Juin 1998.

[Sun90] V. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency Practice and Experience*, 2(4), 1990.

[VV01] S. Vajapeyam et M. Valero. Early 21st Century Processors. *Computer*, 34(4):47–50, Avril 2001.

[WSS93] G.V. Wilson, J. Schaeffer, et D. Szafron. Enterprise in Context: Assessing the Usability of Parallel Programming Environments. In *IBM CASCON '93*, pages 999–1010, 1993.

Bibliographie sur SKiPPER-II

Publications

[Pub1] R. Coudarcher, J. Sérot, et J.-P. Dérutin. Mise en œuvre statique de squelettes de parallélisation dynamiques sous SynDEx. In *$5^{ème}$ Workshop AAA sur l'Adéquation Algorithme Architecture*, pages 189–196, INRIA de Rocquencourt, Janvier 2000. INRIA.

[Pub2] R. Coudarcher, J. Sérot, et J.-P. Dérutin. Implementation of a Skeleton-based Parallel Programming Environment Supporting Arbitrary Nesting. In *F. Mueller, éditeur, 6th International Workshop on High-level Parallel Programming Models and Supportive Environments*, volume LNCS 2026, pages 71–85. Springer. HIPS'01, IPDPS International Symposium, San Francisco, Californie, USA, Avril 2001.

[Pub3] R. Coudarcher, F. Duculty, J. Sérot, F. Jurie, J.-P. Dérutin, et M. Dhome. Parallelisation of a Face Tracking Algorithm with the SKiPPER-II Parallel Programming Environment. Soumis à IEEE CAMP 2002 International Workshop on Computer Architectures for Machine Perception, Indian Institute of Technology Bombay, Mumbai, Inde.

[Pub4] R. Coudarcher, F. Duculty, J. Sérot, F. Jurie, J.-P. Dérutin, et M. Dhome. Skeleton-based Parallelisation of Vision Algorithms on a Beowulf Architecture with the SKiPPER-II Environment. Soumis à IEEE IPDPS 2003 International Parallel and Distributed Processing Symposium, Centre de Convention Nice Acropolis, Nice.

Rapports techniques

[Rap1] A. Béthune, et N. Naudier. Développement d'un serveur de requêtes gérant le bus PCI pour la machine parallèle TransAlpha. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Mars 2000. Rapport de projet de deuxième année d'élève-ingénieur ISIMA.

[Rap2] Q. Guzel, et J. Tiré. Implantation d'une interface MPI sur une machine parallèle MIMD-DM dédiée à la vision artificielle temps réel. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Mars 2000. Rapport de projet de deuxième année d'élève-ingénieur ISIMA.

[Rap3] A. Beuraud, et Y. Elguetioui. Intégration du standard IDL dans l'environnement de programmation parallèle SKiPPER-II. Rapport technique, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Mars 2001. Rapport de projet de deuxième année d'élève-ingénieur ISIMA.

[Rap4] R. Coudarcher. TRACS Project Report: Implementation of a Skeleton-based Parallel Programming Environment Supporting Arbitrary Nesting. *Rapport d'activités pour le programme Européen TRACS (TRaining and Research on Advanced Computing)*, Visite de recherche, EPCC, Ecosse, UK, Mars 2001.

[Rap5] R. Coudarcher, et F. Duculty. Algorithmic Skeleton Nesting with SKiPPER-II: A Case Study. Rapport de recherche, LASMEA, Université Blaise-Pascal, Clermont-Ferrand, Août 2002. Rapport interne RR-G0802/1.

Exposés

[Exp1] R. Coudarcher. *Mise en œuvre statique de squelettes de parallélisation dynamiques sous SynDEx*, 28 janvier 2000, $5^{ème}$ Workshop AAA sur l'Adéquation Algorithme Architecture - AAA 2000, INRIA, Rocquencourt.

[Exp2] R. Coudarcher. *Prise en compte des aspects compositionnels au sein d'une méthodologie de programmation parallèle fondée sur les squelettes fonctionnels.*, 9 mars 2000, Journée de l'école doctorale «Sciences pour l'Ingénieur», Université Blaise-Pascal, Clermont-Ferrand.

[Exp3] R. Coudarcher, et J. Sérot. *Support de la composition de squelettes algorithmiques dans l'environnement de programmation parallèle SKiPPER*, 26 octobre 2000, $3^{ème}$ réunion plénière du GT 7 / GDR-PRC ISIS (AAA - Adéquation Algorithme Architecture), ENST, Paris.

[Exp4] R. Coudarcher. *SKiPPER-II: A Skeleton-based Parallel Programming Environment Supporting Arbitrary Nesting*, 21 février 2001, DSG Seminar, Department of Computing and Electrical Engineering, Université Heriot-Watt, Edimbourg, Ecosse, UK.

[Exp5] R. Coudarcher. *A Brief Overview of the SKiPPER-II Skeleton-based Parallel Programming Environment*, 9 mars 2001, SCOFPIG Meeting, School of Computer Science, Université de St-Andrews, St-Andrews, Ecosse, UK.

[Exp6] R. Coudarcher. *Implementation of a Skeleton-based Parallel Programming Environment Supporting Arbitrary Nesting*, 23 avril 2001, 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS'01, IPDPS International Symposium, San Francisco, Californie, USA.

Annexe A

Rudiments et éléments de syntaxe en Caml pour la sémantique déclarative des squelettes

Un bref exposé des rudiments de Caml nécessaires à la compréhension des notations utilisées dans ce document est fait ici.

A.1 Introduction

Les langages dits *fonctionnels* comme Caml reposent sur la notion de fonction au sens mathématique du terme. Un programme est alors une collection de définitions de fonctions. Chaque appel à l'une d'entre elles déclenche son évaluation. Ces langages permettent donc d'exprimer un ensemble d'expressions mathématiques traduisant les dépendances fonctionnelles.

Grâce à la propriété de *tranparence référentielle* (qui rend la valeur d'une expression indépendante de l'ordre d'évaluation de ses composantes (propriété de Church-Rosser)) les langages fonctionnels sont des formalismes privilégiés pour l'expression du parallélisme. Les squelettes ont la possibilité de s'exprimer facilement dans ces langages sous la forme de fonctions d'ordre supérieur.

Nous présentons ici quelques rudiments de Caml [1] [INR02] afin d'expliciter les notations que nous utilisons pour la sémantique déclarative [2] des squelettes de SKiPPER.

On note dans tout le document par `>...` *les phrases entrées par l'utilisateur, et par* `# ...` *les réponses du compilateur Caml.*

A.2 Types

Il existe trois catégories de types de données en Caml :

- types de base (`int`, `float`, `string`,...),
- type construits, ou composés (`int -> int`, `int list`, `int * int`,...),
- variables de type (`'a`).

Dans la deuxième catégorie on trouve notamment :

- le type $->$ qui permet de créer le type $t1 \rightarrow t2$ à partir des types $t1$ et $t2$, c'est le type de fonctions de domaine $t1$ et de co-domaine $t2$,
- le type `t1 * t2` qui correspond aux couples dont le premier élément a pour type $t1$ et le second $t2$,
- le type `t1 list` qui correspond à une liste d'éléments de type $t1$.

Les variables de type permettent d'exprimer le polymorphisme. Le polymorphisme est une notion qui permet à un objet (variable, fonction,...) d'accepter plusieurs types et donc de pouvoir s'appliquer différemment selon le type qui lui est attribué (la notation $'a$ signifie : «quel que soit le type de a»).

A.3 Définitions

Une définition permet de nommer une valeur ou une fonction.

Elle est réalisée par le mot-clé : `let`.

Par exemple, `let x = 13` associe à l'identificateur x la valeur 13, et `let double x = x * 2` définie une fonction $double$ qui multiplie par deux son argument.

1. Ce langage a ´et´e d´evelopp´e `a l'INRIA dans le cadre du projet *Formel* en 1984. C'est un langage de la famille de ML [GMW79] [Mil84].

2. cf. la notion de s´emantique d´eclarative page 56.

A.4 Listes

La construction de listes d'éléments (éventuellement vide) se fait avec les constructeurs suivants :

- `[]` : signifie une liste vide,
- `e::l` construit une liste non vide avec comme premier élément e, le reste de la liste étant consistué par l.

A.5 Fonctions d'ordre supérieur

Les fonctions d'ordre supérieur sont des fonctions dont les arguments, ou les résultats, sont eux-mêmes des fonctions. Une telle fonction est aussi appelée *fonctionnelle*. La plupart de ces fonctions sont polymorphes.

Les fonctions suivantes en sont deux exemples (voir les sections suivantes) :

- map applique une même fonction f à tous les éléments d'une liste l. Cette fonctionnelle s'écrit en Caml sous la forme :

```
> let rec map f l = match l with
                      []   -> []
                    | x::r -> f x :: map f r
# map : ('a -> 'b) -> 'a list -> 'b list
```

Exemple : `map double [1, 2, 3] = [2, 4, 6]`
La fonctionnelle map utilise un appel explicite au filtrage par l'intermédiaire du mot-clé `match`. L'interprétation est la suivante : si la liste l est vide alors on renvoit une liste vide, sinon on extrait le premier élément de la liste pour lui appliquer la fonction f, puis on applique récursivement map aux éléments restants (d'où le mot-clé `rec`).

- $foldl$ a pour objet d'appliquer itérativement une fonction *f* à une liste *x*, avec pour valeur initiale *z*.

```
> let rec foldl f z x = match x with
                      []   -> z
                    | x::l -> foldl f ( f z x ) l
# foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Exemple : `foldl (+) 0 [1, 2, 3] = 0+1+2+3 = 6`

A.6 Filtrage : map

Le *filtrage* est le mécanisme permettant l'évaluation d'une fonction en évaluant l'ensemble des équations décrivant son comportement selon la configuration des paramètres d'entrées.

Nous nous intéressons ici à la fonctionnelle map utilisée dans la formulation des squelettes. map applique une même fonction f à tous les éléments d'une liste l. L'expression $map\ f\ [x_1, x_2, \ldots, x_n]$ retourne : $[f\ x_1, f\ x_2, ..., f\ x_n]$.

Cette fonctionnelle est utilisée pour donner la sémantique déclarative des squelettes SCM, DF et TF (voire les sections 2.2.2.1 à 2.2.2.3 à partir de la page 55).

A.7 Application itérative d'une fonction à une liste : $foldl$

Ci-dessous est donnée l'expression de la fonctionnelle $foldl$ dont l'objet est d'appliquer itérativement une fonction *f* à une liste *x*, avec pour valeur initiale *z*. Mathématiquement $foldl$ est définie par la relation suivante : $foldl\ f\ z[x_1, x_2, \ldots, x_n] = (\ldots(f(f\ z\ x_1)\ x_2)\ldots\ x_n)$.

Cette fonctionnelle est utilisée pour donner la sémantique déclarative du squelette DF (voire la section 2.2.2.2 page 58).

Annexe B

Mise en œuvre statique des squelettes dynamiques sous SynDEx

La mise en œuvre d'un schéma de parallélisation (*squelette*) dynamique sous SynDEx, telle qu'une ferme de processeurs, pose le délicat problème de sa représentation en termes de graphe flot de données. Nous proposons une technique pour rendre la définition d'un tel schéma totalement statique, et donc permettre sa spécification complète sous SynDEx [CSD00].

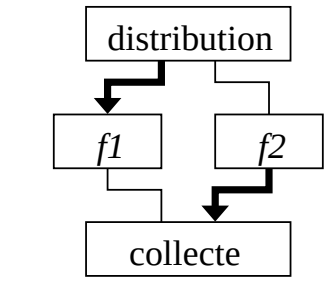
B.1 Formalisation du squelette Data Farming sous forme statique

B.1.1 Squelette DF et formalisme synchrone

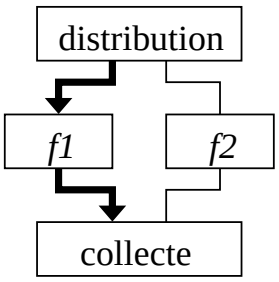
Le squelette DF est un schéma de parallélisation qui met en œuvre un *processus de distribution* des données *et de collecte* des résultats, appelé processus maître, et des *processus de traitements* de ces données, appelés processus esclaves. La particularité de ce schéma est que les données sont distribuées dynamiquement aux esclaves au fur et à mesure que ceux-ci sont capables de les traiter. Une donnée n'est proposée à un esclave que lorsque celui-ci a fini de traiter la donnée précédente. L'avantage de cette technique sur une distribution statique des données (pour laquelle les esclaves reçoivent en bloc l'ensemble des données qu'ils auront à manipuler par la suite) est que la répartition du travail entre les esclaves peut être ajustée à l'exécution en fonction des temps de traitement effectifs sur les esclaves. En d'autres termes, le processus maître gère dynamiquement la répartition de la charge de travail entre les différents esclaves. Cet équilibrage de charge dynamique a de l'intérêt lorsqu'on n'a pas une connaissance préalable sur le nombre de données que l'application aura à manipuler, et/ou lorsque les temps de traitement sont fortement dépendants de ces données.

Nous dirons donc que le fonctionnement de ce squelette est de nature dynamique dans le sens où les communications entre le processus maître et les processus esclaves ne peuvent pas être prédites (et donc ordonnancées) *a priori* à la compilation mais seulement au moment de l'exécution. Par contraste un exécutif distribué comme SynDEx, qui décrit une application sous la forme d'un GFDC, suppose que l'ensemble des communications soient ordonnancées à la compilation.

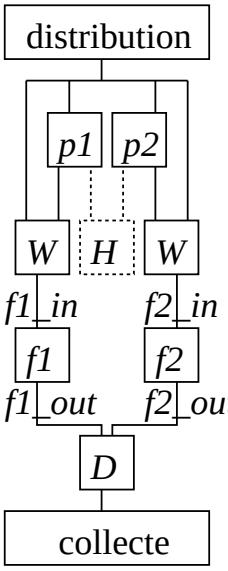
Dans le cadre du squelette DF nous aurions besoin - pour une *même itération* du sous-graphe associé - de pouvoir communiquer des données à *certaines* fonctions de calcul (phase de distribution), et récupérer des résultats en provenance d'*autres* fonctions (phase de collecte). Or cela ne nous est pas permis sous SynDEx (cf. figures B.1a et B.1b). En effet, lorsque une donnée a été transmise à une fonction de calcul, la suite des traitements se situe en sortie de cette fonction, et en aucun cas ailleurs dans le graphe. SynDEx nous permet néanmoins de conditionner l'exécution d'une partie du GFDC. Mais, s'il existe plusieurs chemins entre deux sommets du graphe (par exemple entre une fonction de distribution et une fonction de collecte) et que seulement l'un d'entre eux doit être exécuté à une itération donnée, alors la condition d'exécution s'applique à l'intégralité du chemin concerné, et pas seulement sur le premier arc de transmission des données (cf. figure B.1c). Introduire une seconde condition dans chacun des chemins potentiels pour tenter de les scinder (pour distinguer les entrées des sorties des esclaves par exemple) ne résoud rien puisque cette nouvelle condition sera imbriquée dans la première ; on ne peut pas distinguer les conditions d'entrée et de sortie d'une fonction dans le sens où la condition de sortie est subordonnée à la condition d'entrée. Qui plus est, le conditionnement est fondé sur l'évaluation d'un prédicat. Or si les chemins à exécuter de façon conditionnelle sont répartis sur plusieurs processeurs, la valeur du prédicat doit être distribuée, ce qui engendre une barrière de synchronisation entre les processus. Cette barrière nous interdit de distribuer et de collecter des données sur les esclaves de manière indépendante.



(a) Cheminement souhaite



(b) Cheminement impose par SynDEx



Macro-code type correspondant :

Processeur 0
```
distribution() ; p1() ; p2()
Send(p1) ; Send(p2)
If (p1=true)
   Send(f1_in) ; Recv(f1_out)
EndIf
If (p2=true)
   Send(f2_in) ; Recv(f2_out)
EndIf
collecte()
```

Legende des blocs :
- *W : When*
- *D : Default*
- *H : HAdd*

Processeur 1
```
Recv(p1)
If (p1=true)
   Recv(f1_in)
   f1(f1_in,f1_out)
   Send(f1_out)
EndIf
```

Processeur 2
```
Recv(p2)
If (p2=true)
   Recv(f2_in)
   f2(f2_in,f2_out)
   Send(f2_out)
EndIf
```

distribution(), p1(), p2() et collecte() s'executent sur le processeur 0, f1() sur le processeur 1 et f2() sur le processeur 2

(c) Graphe SynDEx conditionne

FIG. B.1 – *Conditionnement d'un graphe.*

B.1.2 Principe

Pour permettre une description synchrone (statique) du DF, il faut abandonner la définition classique, à savoir un *processus* maître qui distribue à des *processus* esclaves (réalisant le travail) les données dont ils ont besoin au fur et à mesure de leur disponibilité.

Dans un formalisme synchrone, les fonctions de calcul formant les nœuds du graphe flot de données sont supposées avoir une durée d'exécution nulle, leur rôle étant de modifier l'état d'une variable [1].

Les fonctions de calcul que nous utiliserons devraient donc avoir un temps d'exécution pratiquement nul. Or cette contrainte ne peut être imposée. L'idée est donc de faire en sorte, non pas que les fonctions elles-mêmes prennent peu de temps, mais plutôt que le processus maître qui contrôle le DF soit en mesure de s'informer régulièrement de l'état d'avancement des calculs pour chaque esclave.

1. Ou plus pr´ecis´ement la valeur v´ehicul´ee avec l'horloge du signal traversant la fonction.

Pour ce faire, on représente sous SynDEx le squelette DF comme un SCM. La seule différence est qu'on fait apparaître une mémoire d'état [2] liant la distribution des données aux esclaves et leur collecte.

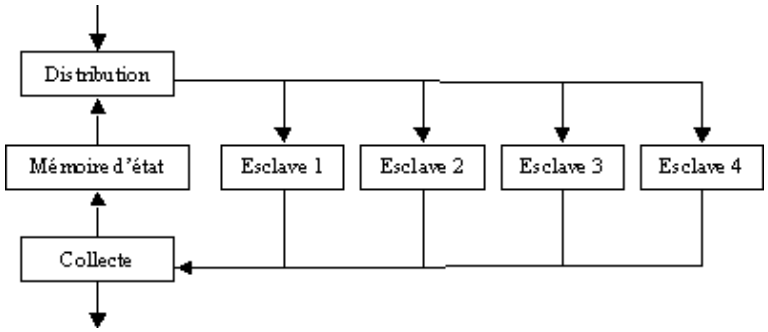


FIG. B.2 – *Représentation du squelette Data Farming statique.*

La fonction de calcul de chaque esclave prenant un certain temps pour s'exécuter, le graphe de la figure B.2 ne représente pas tel quel un DF dans la mesure ou nous sommes obligés d'attendre la fin de *tous* les esclaves avant de pouvoir redistribuer une donnée quelconque à l'un au moins d'entre eux (communications statiques et synchronisme obligent). Pour s'en affranchir, nous proposons non pas d'exécuter la fonction de calcul de chaque esclave entièrement, mais en partie seulement, c'est-à-dire pendant un quota de temps donné (mécanisme similaire à celui utilisé dans les systèmes d'exploitation pour le temps partagé). Ainsi régulièrement la fonction de collecte des résultats du graphe peut s'informer de manière synchrone de l'état d'avancement des calculs de tous les esclaves et indiquer à la fonction de distribution les esclaves ayant terminés et auquels il convient de fournir de nouvelles données.

On exécute donc en boucle le graphe flot de données du DF jusqu'à ce que toutes les données initiales aient été traitées par les esclaves. Grâce à l'exécution par quota de temps des esclaves, le DF peut être décrit sous SynDEx sans qu'il y ait attente de fin d'exécution d'une fonction de calcul assignée à un esclave pour pouvoir en alimenter un autre.

On peut donc considérer le squelette DF ainsi décrit comme un squelette SCM à grain fin itéré.

B.1.3 Implantation

Le mécanisme de «découpage temporel» des fonctions de calcul sur chaque esclave est le suivant.

Initialement, la fonction de distribution émet des données vers tous les esclaves. Ils commencent alors leur travail pour une durée déterminée (quota de temps), en général inférieure à celle nécessaire pour obtenir le résultat. Une fois ce quota épuisé, toutes les fonctions transmettent à la fonction de collecte des résultats des informations indiquant si leur traitement est terminé ou non. La fonction de distribution réémet alors systématiquement (communications statiques et synchronisme obligent) des données vers tous les esclaves. Ces données sont, soit des informations indiquants à l'esclave de reprendre son travail là où il l'a laissé, soit de nouvelles données à traiter pour les esclaves qui avaient terminés.

2. En effet nous verrons que nous nous pla¸cons dans le cadre d'un sch´ema (localement) it´eratif.

Techniquement, ce découpage temporel est fait ainsi :

- sur chaque processeur exécutant un esclave, un gestionnaire d'interruption activé par une horloge est installé ; il permet d'interrompre l'exécution des fonctions de calcul à chaque occurence d'un signal périodique ;
- à chaque fois que la main est redonnée à une fonction de calcul (après réception des données transmises par la fonction de distribution), celle-ci restaure l'état dans lequel elle avait été interrompue et reprend son exécution au niveau de ce point d'arrêt «flottant» ;
- lorsque la routine d'interruption est déclenchée, elle prend soin, avant de rendre la main à la fonction de collecte des résultats, de sauvegarder l'état dans lequel se trouve la fonction de calcul.

Schématiquement on peut représenter le déroulement du DF comme décrit par la figure B.3 (pour deux processeurs (`ROOT` et `PROC`) et pour trois données à traiter (`x1`, `x2` et `x3`)).

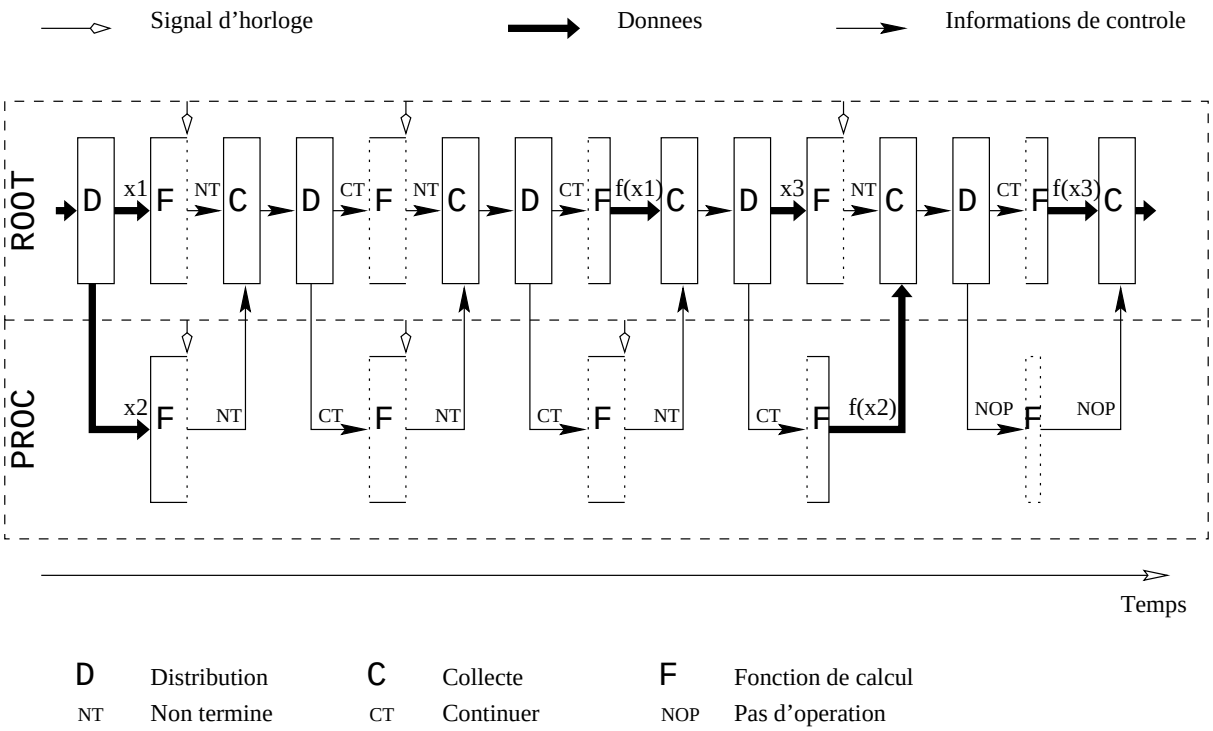


FIG. B.3 – *Synoptique du squelette Data Farming sous forme statique.*

La réalisation de ce schéma suppose l'identification, dans le graphe flot de données de l'application, d'actions (les fonctions D, C et F ainsi que les communications associées) qui sont répétées un certain nombre de fois (dépliage temporel [3]). Cela n'est pas encore disponible en standard sous SynDEx (*v4.3d*); il sera donc nécessaire de mettre en place un mécanisme d'itération sur une partie du graphe sous forme de macros de boucle dans le macro-code généré par SynDEx.

3. Factorisation temporelle *finie.*

B.2 Mise en œuvre

B.2.1 Description sous SynDEx 4.3d

Le DF peut être décrit complètement sous SynDEx à l'exception du mécanisme d'itération qui doit opérer sur ses fonctions (cf. section B.2.2).

Il est représenté comme un SCM, avec en plus une mémoire d'état (M). Cette dernière sert à la fonction *collecte* (C) pour indiquer à la fonction *distribution* (D) quels esclaves (E) ont terminé leur traitement (cf. figure B.4).

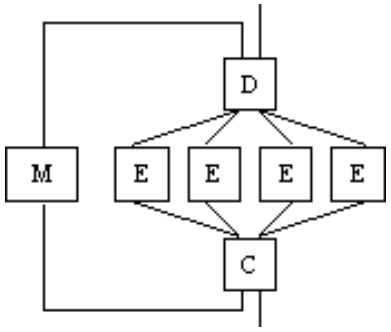


FIG. B.4 – *Description sous forme d'un graphe flot de données.*

B.2.2 Modification du macro-code généré

Au macro-code (fichiers `M4`) généré par SynDEx, il faut rajouter :

– la programmation du timer,

– les routines de gestion des interruptions, de sauvegarde et de restauration de contexte,

– des macros de boucle pour répéter le schéma de la section précédente autant de fois que nécessaire jusqu'à la fin du DF.

C'est ce dernier point qui est le plus délicat car il demande une analyse particulière de ce fichier afin de localiser les éléments (fonctions, mémoires, communications,…) qui font partie d'une entité à itérer. Ces éléments seront fournis en amont par le programmeur, ou par un générateur automatique de code SynDEx.

B.2.3 Routines de gestion en langage C

Les routines gérant le DF sont :

- une fonction de distribution,
- une fonction de collecte,
- une fonction de gestion du contexte d'exécution qui permet de poursuivre une fonction de calcul au point où elle avait été interrompue et d'activer ou de désactiver le gestionnaire d'interruption,
- un gestionnaire d'interruption activé sur un signal d'horloge pour la mise en œuvre du quota de temps alloué à chaque fonction de calcul et la sauvegarde de contexte.

Ces deux dernières routines sont dépendantes de la machine et du système d'exploitation.

Annexe C

Primitives MPI nécessaires au support du noyau de SKiPPER-II sur une machine dédiée au traitement d'images

Sont répertoriées dans cette annexe la liste des primitives MPI nécessaires au support du noyau de SKiPPER-II, et que nous avons implantées dans le cadre du développement d'un sous-ensemble de MPI sur la machine parallèle dédiée au traitement d'images TransAlpha.

C.1 Introduction

Parmi les plate-formes sur lesquelles a fonctionné SKiPPER-II, se trouve la machine dédiée au traitement d'images TransAlpha [BN00]. Cette machine est une machine parallèle de type MIMD-DM architecturée autour d'un réseau (à 100 Mbits/s) de nœuds de calcul constitués par un couple de processeurs Inmos T9000E à 25 MHz (dédié au communications) / DEC Alpha AXP 21066A à 233 MHz [PAR96b] [PAR96a]. Afin de disposer de l'environnement de programmation SKiPPER-II sur cette machine, nous avons été conduits à développer un sous-ensemble du standard MPI. C'est pourquoi nous avons mis en œuvre un jeu restreint de primitives de MPI. Le choix de ces primitives devait permettre à la fois le support du noyau K/II, mais aussi d'anticiper l'utilisation de MPI sur cette machine en implantant d'autres fonctions plus évoluées comme des communications collectives.

C.2 Jeu de primitives MPI retenues pour la machine TransAlpha

Le tableau C.1 récapitule les primitives MPI implantées sur la machine TransAlpha [GT00], classées par catégories, et mentionnant celles effectivement nécessaires au support de SKiPPER-II, les autres ayant été implantées pour d'autres usages.

Primitive	**Objet**	**K/II**
Environnement		
MPI_Init	Active l'environnement	√
MPI_Finalise	Désactive l'environnement	√
MPI_Type_size	Taille du type spécifié	√
MPI_Wtime	Date	√
Contexte		
MPI_Comm_rank	Rang du processus	√
MPI_Comm_size	Nombre de processus	√
MPI_Test	Test de la fin d'un service (non-bloquant)	
MPI_Wait	Test de la fin d'un service (bloquant)	
Communications point-à-point		
MPI_SSend	Envoi de données (bloquant)	√
MPI_Recv	Réception de données (bloquant)	√
MPI_ISend	Envoi de données (non-bloquant)	
Communications collectives		
MPI_Barrier	Synchronisation de processus	
MPI_Bcast	Broadcast	
MPI_Scatter	Distribution sélective	
MPI_Gather	Réception multi-sources	

TAB. C.1 – *Primitives MPI retenues pour la machine TransAlpha (d'après [GT00]).*

Annexe D

Extraits de l'application de multiplication matricielle

Nous proposons dans cette annexe des extraits du code C de l'application de multiplication matricielle sur des entiers de longueur arbitraire (proposée au chapitre 5), et cela à des fins d'illustration des descriptions données sur le fonctionnement et l'utilisation de SKiPPER-II.

D.1 Introduction

Les deux sections qui suivent présentent les extraits des principaux fichiers en langage C formant le code de l'application de multiplication matricielle sur des entiers de longueur arbitraire.

La première section reprend le fichier contenant le codage de la représentation intermédiaire utilisée par SKiPPER-II pour ordonner et gérer les squelettes de l'application.

La seconde donne un aperçu du contenu du fichier regroupant les fonctions de calcul de l'utilisateur et le code nécessaire pour faire la liaison entre celles-ci et le format attendu par SKiPPER-II («stub-code»). Le lecteur notera que dans ces fichiers, *aucun* appel à une fonction quelconque de communication de la bibliothèque MPI n'est fait. Ce fait illustre la gestion complète par le noyau des communications entre les fonctions de calcul de l'utilisateur, et la décharge totale de ce dernier de toutes les tâches bas-niveau de programmation parallèle.

D.2 Fichier codant la représentation intermédiaire de l'application

Le lecteur remarquera le format des arguments des fonctions d'interface avec les fonctions de calcul. Il est conçu pour accepter n'importe quels type et nombre de paramètres au niveau de ces dernières.

Les tableaux nommés *points d'entrée* dans le code permettent au noyau d'appeler les fonctions d'interface lorsque l'application l'impose. Le fait d'utiliser un niveau d'indirection (par «pointeur de fonction» en C) autorise l'accès par le noyau à n'importe quelle fonction de calcul du code utilisateur sans devoir être modifié lors d'un changement de nom ou de format d'une fonction utilisateur. Cela autorise aussi l'emploi de n'importe quel nom de fonction par l'utilisateur, et ne nécessite pas non plus de recompilation du noyau. Seul le fichier codant pour la représentation intermédiaire (et donc le fichier de «liaison») nécessite une recompilation.

Notes :

- la mention `UNDEFINED` pour une entrée signifie qu'elle est sans objet,
- `END_OF_APP` indique que le squelette est le dernier de l'application,
- `NO_TARGET` a pour rôle d'informer le noyau que le squelette courant est le dernier squelette de la hiérarchie d'imbrication, et donc, que les résultats de celui-ci doivent être remontés vers le squelette le plus englobant,
- `MASTER` signifie que la fonction de calcul est en réalité un squelette imbriqué,
- `SLAVE` indique que le noyau doit utiliser la fonction utilisateur correspondant au squelette comme fonction de calcul.

```
/* DECLARATION DES FONCTIONS D'INTERFACE */

/* Prototypes des fonctions SPLIT */
int matrix_split_outter( char *, int, int **, int * );
int matrix_split_inner( char *, int, int **, int * );

/* Prototypes des fonctions PREDICATE */
int matrix_predicate( char *, int, int * );

/* Prototypes des fonctions COMPUTE */
int matrix_compute_inner( char *, int, char **, int * );

/* Prototypes des fonctions DIVIDE */
int matrix_divide( char *, int, char **, int *, int **, int * );

/* Prototypes des fonctions MERGE */
int matrix_merge_outter( char *, int, char **, int * );
int matrix_merge_inner( char *, int, char **, int * );

/* NOMBRE TOTAL DE SQUELETTES */
#define SQL_NUMBER  2

/* POINTS D'ENTREE DES FONCTIONS D'INTERFACE */

/* Fonctions COMPUTE */
int (* kd_slaves[SQL_NUMBER])
  ( char *  buffer_in_data ,
    int     buffer_in_size ,
    char ** buffer_out_data,
    int  *  buffer_out_size
  ) =
{
  UNDEFINED,
  &matrix_compute_inner
};

/* Fonctions SPLIT */
int (* kd_split[SQL_NUMBER])
  ( char *  buffer_in_data,
    int     buffer_in_size,
    int  ** offsets_buffer,
    int  *  offsets_size
  ) =
{
  &matrix_split_outter,
  &matrix_split_inner
};

/* Fonctions PREDICATE */
int (* kd_predicate[SQL_NUMBER])
  ( char *  buffer_in_data,
    int     buffer_in_size,
    int  *  solve
  ) =
{
  UNDEFINED,
  &matrix_predicate
};
```

```
/* Fonctions DIVIDE */
int (* kd_divide[SQL_NUMBER])
  ( char *  buffer_in_data ,
    int     buffer_in_size ,
    char ** buffer_out_data,
    int  *  buffer_out_size,
    int  ** offsets_buffer ,
    int  *  offsets_size
  ) =
{
  UNDEFINED,
  &matrix_divide
};

/* Fonctions MERGE */
int (* kd_merge[SQL_NUMBER])
  ( char *  buffer_slave_recv_data,
    int     buffer_slave_recv_size,
    char ** buffer_out_data       ,
    int  *  buffer_out_size
  ) =
{
  &matrix_merge_outter,
  &matrix_merge_inner
};

/* DESCRIPTEUR DE LA STRUCTURE DE L'APPLICATION */
Jump kd_jump[SQL_NUMBER] =
{
  {END_OF_APP, MASTER, 1         },
  {NO_TARGET , SLAVE , UNDEFINED }
};
```

D.3 Fonctions de calcul de l'utilisateur, et code de liaison

```
#define MATRIX_SIZE             25

#define MATRIX_A_ROW_SIZE       MATRIX_SIZE
#define MATRIX_A_COLUMN_SIZE    MATRIX_SIZE
#define MATRIX_B_ROW_SIZE       MATRIX_A_COLUMN_SIZE
#define MATRIX_B_COLUMN_SIZE    MATRIX_SIZE
#define ALI_SIZE                50


char matrix_a[MATRIX_A_ROW_SIZE][MATRIX_A_COLUMN_SIZE][ALI_SIZE];
char matrix_b[MATRIX_B_ROW_SIZE][MATRIX_B_COLUMN_SIZE][ALI_SIZE];
char matrix_r[MATRIX_A_ROW_SIZE][MATRIX_B_COLUMN_SIZE][ALI_SIZE];

char ali[ALI_SIZE];


/* FONCTIONS DE CALCUL DE L'UTILISATEUR */

int addc( char pos, char carry, char * result )
{
  int     d;
  char    rt;


  for( d=pos; d<ALI_SIZE; d++ )
  {
    rt = result[d];
    result[d] = (rt+carry) % 10;
    carry     = (rt+carry) / 10;
  }
}
```

```
int add( char carry, char * ali1, char * ali2, char * result )
{
  int    d, a;


  memcpy( result, ali1, ALI_SIZE );
  addc( 0, carry, result );
  for( d=0; d<ALI_SIZE; d++ )
  {
    addc( d, ali2[d], result );
  }
}


int multd( char digit, char * ali, char * result )
{
  int    a, d, r;
  char   carry = 0;


  for( d=0; d<ALI_SIZE; d++ )
  {
    r         = digit*ali[d];
    result[d] = (r+carry) % 10;
    carry     = (r+carry) / 10;
  }
}


int mult( char * ali1, char * ali2, char * result )
{
  int    a, d;
  char   local_ali1[ALI_SIZE], local_ali2[ALI_SIZE*2];
  char  * buffer1;


  for( a=0; a<ALI_SIZE; a++)
  {
    local_ali2[a] = 0;
  }
  buffer1 = local_ali2+ALI_SIZE;
  memcpy( buffer1, ali2, ALI_SIZE);

  for( d=0; d<ALI_SIZE; d++ )
  {
    multd( ali1[d], buffer1, local_ali1 );
    add( 0, result , local_ali1, result );
    buffer1--;
  }
}

int innerprod( char vect1[MATRIX_A_COLUMN_SIZE][ALI_SIZE],
               char vect2[MATRIX_A_COLUMN_SIZE][ALI_SIZE],
               char * result )
{
  int    v, a;
  char   local_ali[ALI_SIZE];


  for( v=0; v<MATRIX_A_COLUMN_SIZE; v++ )
  {
    for( a=0; a<ALI_SIZE; a++)
    {
      local_ali[a] = 0;
    }
    mult( vect1[v], vect2[v], local_ali );
    add( 0, result, local_ali, result );
  }
}
```

```
/* FONCTIONS D'INTERFACE AVEC LES FONCTIONS DE CALCUL */

/* Fonction initiale, permettant par exemple l'acquisition des donnees */
int app_begin( char ** buffer_out_data, int * buffer_out_size )
{
  int     data_size;
  int     r1, c2, v, a;
  char  * buffer;


  data_size   =   MATRIX_A_ROW_SIZE*MATRIX_B_COLUMN_SIZE
                * ((MATRIX_A_COLUMN_SIZE+MATRIX_B_ROW_SIZE)*ALI_SIZE+2);

  *buffer_out_size = data_size;
  *buffer_out_data = (char *) malloc( *buffer_out_size );

  buffer = *buffer_out_data;
  for( r1=0; r1<MATRIX_A_ROW_SIZE; r1++ )
  {
    for( c2=0; c2<MATRIX_B_COLUMN_SIZE; c2++ )
    {
      *buffer = (char) r1; buffer++;
      *buffer = (char) c2; buffer++;
      for( v=0; v<MATRIX_A_COLUMN_SIZE; v++ )
      {
        for( a=0;          a<ALI_SIZE/2; a++ ) buffer[v*ALI_SIZE+a] = rand() % 10;
        for( a=ALI_SIZE/2; a<ALI_SIZE;   a++ ) buffer[v*ALI_SIZE+a] = 0;
      }
      buffer += MATRIX_A_COLUMN_SIZE*ALI_SIZE;
      for( v=0; v<MATRIX_B_ROW_SIZE; v++ )
      {
        for( a=0;          a<ALI_SIZE/2; a++ ) buffer[v*ALI_SIZE+a] = rand() % 10;
        for( a=ALI_SIZE/2; a<ALI_SIZE;   a++ ) buffer[v*ALI_SIZE+a] = 0;
      }
      buffer += MATRIX_A_COLUMN_SIZE*ALI_SIZE;
    }
  }

  puts("BEGIN");
  fflush( stdout );

  return NO_ERROR;
}

/* Fonction terminale de l'application, permettant la restitution des resultats */
int app_end( char * buffer_in_data, int buffer_in_size )
{
  int    r, c, a;


  for( r=0; r<MATRIX_A_ROW_SIZE; r++ )
  {
    for( c=0; c<MATRIX_B_COLUMN_SIZE; c++ )
    {
      for( a=0; a<ALI_SIZE; a++ )
      {
        printf( "%u", matrix_r[r][c][a] );
      }
      printf( " " );
    }
    printf( "\n" );
  }

  puts("END");
  fflush( stdout );

  return NO_ERROR;
}
```

```
int  matrix_split_outter( char *  buffer_in_data,
                          int     buffer_in_size,
                          int  ** offsets_buffer,
                          int  *  offsets_size    )
{
  int    i;


  *offsets_size = MATRIX_A_ROW_SIZE +1;
  *offsets_buffer
    = (int *) malloc( *offsets_size *sizeof( int ) );
  for( i=0; i<*offsets_size; i++ )
  {
    (*offsets_buffer)[i] = i*(MATRIX_A_COLUMN_SIZE*2*ALI_SIZE +2)*MATRIX_B_COLUMN_SIZE;
  }

  return NO_ERROR;
}

int  matrix_merge_outter( char *  buffer_in_data,
                          int     buffer_in_size,
                          char ** buffer_out_data,
                          int  *  buffer_out_size  )
{
  static  int    call = 0;
  char        *  buffer;


  *buffer_out_size += MATRIX_B_COLUMN_SIZE*(ALI_SIZE+2);
  *buffer_out_data
      = (char *) realloc( *buffer_out_data,
                          *buffer_out_size *sizeof( char ) );

  for( buffer=buffer_in_data; buffer<buffer_in_data+buffer_in_size; buffer+=ALI_SIZE+2 )
  {
    memcpy( &(matrix_r[*buffer][*(buffer+1)]), buffer+2, ALI_SIZE );
  }
  call++;

  return NO_ERROR;
}

int  matrix_compute_inner( char *  buffer_in_data,
                           int     buffer_in_size,
                           char ** buffer_out_data,
                           int  *  buffer_out_size  )
{
  int     c;
  char    (*vect1)[MATRIX_A_COLUMN_SIZE][ALI_SIZE];
  char    (*vect2)[MATRIX_A_COLUMN_SIZE][ALI_SIZE];


  *buffer_out_size = ALI_SIZE+2;
  *buffer_out_data = (char *) malloc( *buffer_out_size );
  (*buffer_out_data)[0] = *buffer_in_data;
  (*buffer_out_data)[1] = *(buffer_in_data+1);
  for( c=2; c<*buffer_out_size; c++ ) (*buffer_out_data)[c] = 0;

  vect1 = buffer_in_data+2;
  vect2 = buffer_in_data+2+(MATRIX_A_COLUMN_SIZE*ALI_SIZE);

  /* Appel effectif de la fonction de calcul de l'utilisateur */
  innerprod( *vect1, *vect2, (*buffer_out_data)+2 );

  return NO_ERROR;
}
```

```
int  matrix_split_inner( char *  buffer_in_data,
                         int     buffer_in_size,
                         int  ** offsets_buffer,
                         int  *  offsets_size    )
{
  int    i;


  *offsets_size = MATRIX_B_COLUMN_SIZE +1;
  *offsets_buffer
    = (int *) malloc( *offsets_size *sizeof( int ) );
  for( i=0; i<*offsets_size; i++ )
  {
    (*offsets_buffer)[i] = i*(MATRIX_A_COLUMN_SIZE*2*ALI_SIZE +2);
  }

  return NO_ERROR;
}

int  matrix_merge_inner( char *  buffer_in_data,
                         int     buffer_in_size,
                         char ** buffer_out_data,
                         int  *  buffer_out_size  )
{
  static  int    call = 0;


  if (!call)
  {
    *buffer_out_size = buffer_in_size*MATRIX_B_COLUMN_SIZE;
    *buffer_out_data = (char *) malloc( *buffer_out_size *sizeof( char ) );
  }

  memcpy( (*buffer_out_data)+(buffer_in_size*call), buffer_in_data, buffer_in_size );
  call++;

  return NO_ERROR;
}

int  matrix_predicate( char *  buffer_in_data,
                       int     buffer_in_size,
                       int  *  solve           )
{
  /* Donnees pretes a etre traitees */
  *solve = YES_;

  return NO_ERROR;
}

int  matrix_divide( char *  buffer_in_data,
                    int     buffer_in_size,
                    char ** buffer_out_data,
                    int  *  buffer_out_size,
                    int  ** offsets_buffer,
                    int  *  offsets_size     )
{
  /* Sans objet */
  return NO_ERROR;
}
```

Annexe E

Exemple de programme écrit avec SKiPPER-II, et directement en C et MPI

Nous proposons dans cette annexe l'exemple d'un même programme écrit sous deux formes différentes, l'une pour l'environnement SKiPPER-II et l'autre directement en C avec la bibliothèque MPI, à des fins de comparaison. Ce programme réalise un simple calcul de type multiplication dans les esclaves de deux SCM imbriqués.

E.1 Version SKiPPER-II

E.1.1 Fichier codant la représentation intermédiaire de l'application

Le lecteur remarquera le format des arguments des fonctions d'interface avec les fonctions de calcul. Il est conçu pour accepter n'importe quels type et nombre de paramètres au niveau de ces dernières.

Les tableaux nommés *points d'entrée* dans le code permettent au noyau d'appeler les fonctions d'interface lorsque l'application l'impose. Le fait d'utiliser un niveau d'indirection (par «pointeur de fonction» en C) autorise l'accès par le noyau à n'importe quelle fonction de calcul du code utilisateur sans devoir être modifié lors d'un changement de nom ou de format d'une fonction utilisateur. Cela autorise aussi l'emploi de n'importe quel nom de fonction par l'utilisateur, et ne nécessite pas non plus de recompilation du noyau. Seul le fichier codant pour la représentation intermédiaire (et donc le fichier de «liaison») nécessite une recompilation.

Notes :

- la mention `UNDEFINED` pour une entrée signifie qu'elle est sans objet,
- `END_OF_APP` indique que le squelette est le dernier de l'application,
- `NO_TARGET` a pour rôle d'informer le noyau que le squelette courant est le dernier squelette de la hiérarchie d'imbrication, et donc, que les résultats de celui-ci doivent être remontés vers le squelette le plus englobant,
- `MASTER` signifie que la fonction de calcul est en réalité un squelette imbriqué,
- `SLAVE` indique que le noyau doit utiliser la fonction utilisateur correspondant au squelette comme fonction de calcul.

```
/* DECLARATION DES FONCTIONS D'INTERFACE */

/* Prototypes des fonctions SPLIT */
int scm1_rowblock( char *, int, int **, int * );
int scm2_rowblock( char *, int, int **, int * );

/* Prototypes des fonctions PREDICATE */
int scm2_predicate( char *, int, int * );

/* Prototypes des fonctions COMPUTE */
int scm2_comp( char *, int, char **, int * );

/* Prototypes des fonctions DIVIDE */
int scm2_divide( char *, int, char **, int *, int **, int * );

/* Prototypes des fonctions MERGE */
int scm1_fus( char *, int, char **, int * );
int scm2_fus( char *, int, char **, int * );

/* NOMBRE TOTAL DE SQUELETTES */
#define SQL_NUMBER  2
```

```
/* POINTS D'ENTREE DES FONCTIONS D'INTERFACE */

  /* Fonction COMPUTE */
  int (* kd_slaves[SQL_NUMBER])
    ( char *  buffer_in_data  ,
      int     buffer_in_size  ,
      char ** buffer_out_data ,
      int  *  buffer_out_size
    ) =
  {
    UNDEFINED,
    &scm2_comp
  };

  /* Fonction SPLIT */
  int (* kd_split[SQL_NUMBER])
    ( char *  buffer_in_data,
      int     buffer_in_size,
      int  ** offsets_buffer,
      int  *  offsets_size
    ) =
  {
    &scm1_rowblock,
    &scm2_rowblock
  };

  /* Fonction PREDICATE */
  int (* kd_predicate[SQL_NUMBER])
    ( char *  buffer_in_data,
      int     buffer_in_size,
      int  *  solve
    ) =
  {
    UNDEFINED,
    &scm2_predicate
  };

  /* Fonction DIVIDE */
  int (* kd_divide[SQL_NUMBER])
    ( char *  buffer_in_data,
      int     buffer_in_size,
      char ** buffer_out_data,
      int  *  buffer_out_size,
      int  ** offsets_buffer,
      int  *  offsets_size
    ) =
  {
    UNDEFINED,
    &scm2_divide
  };

  /* Fonction MERGE */
  int (* kd_merge[SQL_NUMBER])
    ( char *  buffer_slave_recv_data,
      int     buffer_slave_recv_size,
      char ** buffer_out_data,
      int  *  buffer_out_size
    ) =
  {
    &scm1_fus,
    &scm2_fus
  };

  /* DESCRIPTEUR DE LA STRUCTURE DE L'APPLICATION */
  Jump kd_jump[SQL_NUMBER] =
  {
    {END_OF_APP, MASTER, 1         },
    {NO_TARGET , SLAVE , UNDEFINED }
  };
```

E.1.2 Fonctions de calcul de l'utilisateur, et code de liaison

```
#define K_        *1024

#define DATA_DIM   1000 /* (in Kbytes) */
#define COMP_DIM  10000 /* (in Kflops) */


extern int    kd_proc_nbr;
extern int    kd_proc_num;


/* FONCTIONS D'INTERFACE ET DE CALCUL */

/* Fonction initiale, permettant par exemple l'acquisition des donnees */
int app_begin( char ** buffer_out_data, int * buffer_out_size )
{
  int    i;


  puts( "BEGIN" );

  *buffer_out_size = DATA_DIM K_;
  *buffer_out_data
    = (char *) malloc( *buffer_out_size *sizeof( char ) );

  return NO_ERROR;
}

/* Fonction terminale de l'application, permettant la restitution des resultats */
int app_end( char * buffer_in_data, int buffer_in_size )
{

  puts( "END" );

  return NO_ERROR;
}

int  scm1_rowblock( char *  buffer_in_data,
                    int     buffer_in_size,
                    int  ** offsets_buffer,
                    int  *  offsets_size    )
{
  int    i;


  *offsets_size = 2 +1;
  *offsets_buffer
    = (int *) malloc( *offsets_size *sizeof( int ) );
  for( i=0; i<*offsets_size; i++ )
  {
    (*offsets_buffer)[i] = i*(DATA_DIM K_)/(*offsets_size -1);
  }

  return NO_ERROR;
}
```

```
int  scm1_fus( char *  buffer_in_data,
               int     buffer_in_size,
               char ** buffer_out_data,
               int  *  buffer_out_size  )
{
  int           j, k;
  static int    call_number = 0;


  call_number++;

  *buffer_out_size += buffer_in_size;
  *buffer_out_data
    = (char *) realloc( *buffer_out_data,
                        *buffer_out_size *sizeof( char ) );

  return NO_ERROR;
}

int  scm2_comp( char *  buffer_in_data,
                int     buffer_in_size,
                char ** buffer_out_data,
                int  *  buffer_out_size  )
{
  int      x;
  float    f = 1.17;


  *buffer_out_size = buffer_in_size;
  *buffer_out_data
    = (char *) malloc( *buffer_out_size *sizeof( char ) );

  for( x=0; x<(COMP_DIM K_)/kd_proc_nbr; x++ ) f *= 3.33;

  return NO_ERROR;
}

int  scm2_rowblock( char *  buffer_in_data,
                    int     buffer_in_size,
                    int  ** offsets_buffer,
                    int  *  offsets_size    )
{
  int    i;


  *offsets_size = kd_proc_nbr/2 +1;
  *offsets_buffer
    = (int *) malloc( *offsets_size *sizeof( int ) );
  for( i=0; i<*offsets_size; i++ )
  {
    (*offsets_buffer)[i] = i*(buffer_in_size)/(kd_proc_nbr/2);
  }

  return NO_ERROR;
}
```

```
int  scm2_fus( char *  buffer_in_data,
               int     buffer_in_size,
               char ** buffer_out_data,
               int  *  buffer_out_size  )
{
  int           j, k;
  static int    call_number = 0;


  call_number++;

  *buffer_out_size += buffer_in_size;
  *buffer_out_data
    = (char *) realloc( *buffer_out_data,
                        *buffer_out_size *sizeof( char ) );

  return NO_ERROR;
}

int scm2_predicate( char *  buffer_in_data,
                    int     buffer_in_size,
                    int  *  solve           )
{
  /* Donnees pretes a etre traitees */
  *solve = YES_;

  return NO_ERROR;
}

int scm2_divide( char *  buffer_in_data,
                 int     buffer_in_size,
                 char ** buffer_out_data,
                 int  *  buffer_out_size,
                 int  ** offsets_buffer,
                 int  *  offsets_size     )
{
  /* Sans objet */

  return NO_ERROR;
}
```

E.2 Version C/MPI

```
#define K_        *1024
#define DATA_DIM   1000
#define COMP_DIM  10000

MPI_Comm    kd_comm;
int         kd_rank;
int         kd_size;
int         kd_proc_nbr;
int         kd_proc_num;
char      * buffer;


int main( int argc, char ** argv )
{
  int           i;
  float         f = 1.17;
  MPI_Status    status;

  MPI_Init( &argc, &argv );
  kd_comm = MPI_COMM_WORLD;
  MPI_Comm_size( kd_comm, &kd_size );
  MPI_Comm_rank( kd_comm, &kd_rank );
  kd_proc_nbr=kd_size;
  kd_proc_num=kd_rank;
  if(!kd_proc_num)
  {
    puts( "BEGIN" );

    buffer = (char *) malloc( DATA_DIM K_ );
    for( i=1; i<kd_proc_nbr; i++ )
    {
      MPI_Send( buffer, (DATA_DIM K_)/(kd_proc_nbr-1), MPI_CHAR, i, 0, kd_comm );
    }
    while(--i)
    {
      MPI_Recv( buffer, (DATA_DIM K_)/(kd_proc_nbr-1), MPI_CHAR, MPI_ANY_SOURCE,
                0, kd_comm, &status );
    }
    free( buffer );

    puts( "END" );
  }
  else
  {
    buffer = (char *) malloc( (DATA_DIM K_)/(kd_proc_nbr-1) );
    MPI_Recv( buffer, (DATA_DIM K_)/(kd_proc_nbr-1), MPI_CHAR, 0,
              0, kd_comm, &status );
    for( i=0; i<(COMP_DIM K_)/(kd_proc_nbr-1); i++ ) f *= 3.33;
    MPI_Send( buffer, (DATA_DIM K_)/(kd_proc_nbr-1), MPI_CHAR, 0, 0, kd_comm );
    free( buffer );
  }
  MPI_Finalize();
  return 0;
}
```